Integrating EXPRESS and SGML for Document Modelling in Control Systems Design

Juan Bicarregui and Brian Matthews
Systems Engineering
Rutherford Appleton Laboratory
Chilton, Didcot, OXON, U.K.
{jcb,bmm}@inf.rl.ac.uk

October 19, 1995

Abstract

This paper considers the integration of documents written using the Standard Generalized Markup Language (SGML) into an information modelling context using EXPRESS. An architecture is presented which allows storage and retrieval of SGML Document Type Definitions (DTDs), and documents which conform to those DTDs. Also considered is the extraction of documents from the EXPRESS information model which incorporate values from other, non-documentary information represented in EXPRESS.

1 Introduction

The computing systems for industrial process automation projects are typically complex, distributed, real-time systems. Several different types of information are used in such a project, including data for the components, project documents and process models. A significant proportion of the total design effort in control systems design is in the production of documentation both to accompany the product, and to document the design process. Many of these documents are of standard forms tailored to the details of a particular design.

In order to streamline the production and reuse of such documents, it is widely accepted that they should be developed in a structured format so that generic document schemas and sections of documents can be classified, stored and retrieved by structure, without recourse to an analysis of semantic content. Systematic reuse of this sort would greatly improve productivity. To this end, the ISO sponsored Standard Generalised Markup Language (SGML) [2] is available as a means of developing structured documents.

Nevertheless, to efficiently store and retrieve information used in the design of a control system, and to smoothly integrate one information type with another, it is desirable to use one common representation to store all information types. The EXPRESS language [3] is a widely accepted standard for representing engineering data, and in this paper we describe the use of EXPRESS as the common medium for storage and exchange of all data, including documents. However, it would also be desirable to handle documents in SGML to allow the ease of expression in structuring and writing documents and also specialised document generation tools to be used. To facilitate this, we describe the the provision of translators between EXPRESS and SGML.

This paper describes how document modelling can be integrated into an EXPRESS system. The work has been undertaken in a larger exercise in information structuring in the context of the reuse of information, in the Esprit III project TORUS[1]. In a companion paper in this conference, we discuss how process modelling can be integrated into the same framework [4]. Related work in this area is described in[1, 6].

The next section gives a brief introduction to some of the features of SGML via an example. The section 3 discusses architectural issues arising in integrating EXPRESS and SGML. We then discuss some of the details of these translations: in section 4, that of representing SGML Document Data Types in EXPRESS; in section 5, generating EXPRESS schemas from SGML Document Data Types; and in section 6, extracting SGML documents from EXPRESS whilst incorporating EXPRESS data. Finally we report on the ongoing implementation of this architecture and mention some further areas of investigation.

2 The Standard Generalised Markup Language

SGML provides a standard method of structuring and exchanging documents. It provides a mechanism for defining the textual conventions (character set, language etc) within which a document is written and also allows the definition of a "markup". It is this markup which defines the structure of documents by identifying generic sections of the text of a document, and by specifying which sections can lie within the scope of others. Each document compatible with SGML provides a reference to a *Document Type Definition* (DTD) which defines the markup. DTDs provide the basis of the document structuring which we wish to relate to the information structuring provided by EXPRESS.

Each kind of document used in an organisation can be given as a DTD. These are then interpreted in standard ways to generate the document in the desired format. As an example to illustrate the restricted subset of SGML considered in this paper, we describe the structure of a simple report. The Document Type Definition of such a document is given in Figure 2. Line numbers have been added to left of each line. DTDs comprise a

[1]ESPRIT III project 8080 - TORUS: Tools for Object Based Large Scale Reuse for Industrial Systems Design, a collaboration between Cegelec Projects Ltd (UK), FLS Automation A/S (Denmark), Rutherford Appleton Laboratory (UK) and Alcatel-ISR (France). TORUS seeks to raise the level of reuse in industrial process control projects.

```
(1)   <!--       DTD for simple Report                          --  >
(2)   <!ENTITY   RAL       Rutherford Appleton Laboratory          >
(3)   <!--       ELEMENTS  MIN       CONTENTS                  --  >
(4)   <!ELEMENT  Report    - -       (Head, Body, Appendix?)       >
(5)   <!ELEMENT  Head      - -       (Title, Author)               >
(6)   <!ELEMENT  Title     - O       (#PCDATA)                     >
(7)   <!ELEMENT  Author    - O       (#PCDATA)                     >
(8)   <!ELEMENT  Body      - -       (Section+)                    >
(9)   <!ELEMENT  Section   - -       (H1,P*)                       >
(10)  <!ELEMENT  H1        - O       (#PCDATA)                     >
(11)  <!ELEMENT  P         - O       (#PCDATA | List | Subsect)    >
(12)  <!ELEMENT  Subsect   - -       (H2,Q*)                       >
(13)  <!ELEMENT  H2        - O       (#PCDATA)                     >
(14)  <!ELEMENT  Q         - O       (#PCDATA|List)                >
(15)  <!ELEMENT  List      - -       (L1 & Item+)                  >
(16)  <!ELEMENT  L1        - O       (#PCDATA)                     >
(17)  <!ELEMENT  Item      - O       (#PCDATA)                     >
(18)  <!ELEMENT  Appendix  - -       (A1,P*)                       >
(19)  <!ELEMENT  A1        - O       (#PCDATA)                     >
(20)  <!--       ELEMENTS  NAME      VALUE           DEFAULT --  >
(21)  <!ATTLIST  Report    security  (confiden|public)   public
(22)                       status    (draft|final)        draft
(23)                       refid     CDATA             #REQUIRED  >
```

Figure 1: Document Type Definition of a Simple Report

series of declarations which begin[2] with `<` and end with `>`. There are three major types of declaration in SGML: *elements*, *attributes*, and *entities*[3]. Text, such as lines 1, 3, and 20, contained within a pair of `--` is ignored as comments.

2.1 Elements

Structure is defined in a Document Type Definition by means of *element* declarations, distinguished by the keyword `!ELEMENT`. Such an element declaration consists of an element name, a *minimisation*, and a *group*. The minimisation declares whether the user can omit begin or end tags for this element in documents; it has two characters, each of which can be either "`-`" for retain or "`O`" for omit.

The group declaration defines how the element is constructed. This can be simply text, such as the element `Title` in line 6, denoted by the keyword `#PCDATA` (for Parsed Character DATA), or it can have a more complicated structure referring to other elements. This is defined using a notation similar to that of regular expressions, as described below.

At the top level, a `Report` (line 4) has three components in sequence: a `Head`, a `Body` and an optional `Appendix`, optional since it has a `?` suffix. In line 5, a `Head` is declared to be a `Title` followed by an `Author`, both of which are bottom level `#PCDATA` elements. A

[2]The syntax of SGML is redefinable. In this paper we shall follow the standard reference syntax

[3]It is unfortunate that there is some overlap in the terminology of SGML and EXPRESS, especially in the terms "attribute" and "entity". Where confusion may be caused, we shall refer to "SGML attributes" and "SGML entity", whilst reserving "attributes" and "entity" for EXPRESS.

`Body` (line 8) consists of a non-empty recurrence of `Sections`, defined using the `+` suffix, and in line 9, a `Section` is declared to have a heading `H1` followed by a `P*` representing a recurrence of paragraphs, which could be empty. A paragraph element `P` (line 11) is formed by a choice of elements, separated by `|`s, either plain text (`#PCDATA`), a `List` element, or a subsection, defined by the `Subsect` element. The `List` (line 15) element is a pair of elements one of which is a list title, element `L1`, and the other is a non-empty recurrence of list items `Item`, but these can occur in either order, as denoted by the `&` separating them. Other elements in this DTD follow a similar pattern.

2.2 Attributes

Lines 21-23 declare *attributes*. Attributes are additional non-structural information which can be associated with an element. Such information is not used for structuring by SGML, but can affect the appearance of the document. In this example, three attributes are declared all of which pertain to the element `Report`: `security` which can take the alternative values `confiden` or `public`, so the element can be marked with the level of security of the document; `status` which can take the alternative values `draft` or `final`; and `refid` which is a value of type `CDATA`, that is Character DATA, the SGML base type for text which does not need further processing. Thus this attribute represents an identifier for the document. An attribute also has a default value for cases in which the explicit setting of the attribute is omitted in a document instance. Thus `security` has the default `public`, and status has default `draft`, whereas `refid` has the keyword `#REQUIRED` denoting that it always has to be explicitly set by the document writer.

2.3 Entities

Line 2 gives an example of an SGML *entity*. At their most straightforward, entities provide "macro substitution"; the text assigned to an entity in the declaration is expanded in the text when the entity is referred to. Thus we have an entity `RAL` which will expand out into the text "Rutherford Appleton Laboratory" when an *entity reference* to `RAL` is placed in the text of a document. Entities can be used in a more sophisticated manner, instance for defining non-ASCII characters and alphabets, and also can contain markup which is interpreted on use. For the purposes of this paper, we restrict their use to the one described above.

2.4 Document Instances

The DTD thus defines a template for marking up documents, and any document which conforms to this DTD can only use markup in the way specified by the DTD. A small document which conforms to the example DTD is given in Figure 2.3. The first line uses a `!DOCTYPE` declaration to declare which DTD the document conforms to: this can either be a local declaration, or in this case a reference to a file, *report.dtd*, where the DTD is stored. Each instance of an element is enclosed by a begin tag <*element_name*> and an

```
<!DOCTYPE report "report.dtd" >
<Report status=final refid="RAL/1">
  <Head>
    <Title> A Short Report on the Laboratory
    <Author> Brian Matthews
  </Head>
  <Body>
    <Section>
      <H1> Introduction
      <P> The &RAL; is a research laboratory which forms one of the constituent
          sites of the Central Laboratory of the Research Council ...
      <P> ...
    </Section>
    <Section>
      <H1> Systems Engineering
      <P> ...
    </Section>
  </Body>
</Report>
```

Figure 2: A Simple Report Document

end tag </*element_name*>. Tags can be omitted if the relevant tag minimisation allows. Attributes are also set within the begin tag, so the tag `<Report ...>` contains values for two of its attributes. The missing attribute, `security` is assumed to take its default value, `public`. The entity `RAL` is referred to in the text by an entity reference `&RAL;`. This will thus expand in place to its value.

Note that SGML only defines the structure of documents: it is not concerned with defining their appearance. In order to produce documents in a given document preparation system, a mapping from the DTD to that document preparation system must be defined which specifies the appearance. For example, it is at this stage that it may be decided to generate headings in a larger, bold font, with more line separation.

3 Architecture for Integrating SGML and EXPRESS

To enable the use of SGML to describe and manipulate documents, yet to store, search and retrieve those same documents using EXPRESS, requires an architecture for integrating documents and document models within an EXPRESS framework. In a paper in the 1994 EXPRESS User Group conference [1], a mapping from SGML to EXPRESS was proposed which generates EXPRESS schema from SGML Document Type Definitions (DTDs). This mapping forms the basis of instance translation functions used to store documents which conform to the DTD into EXPRESS instances conforming to the generated schema.

In this paper we extend this approach by also defining DTDs in terms of generic EXPRESS schema, and by a mechanism to insert data from the information model into documents.

We thus integrate the SGML document model and the EXPRESS information model. An architecture which achieves this integration by a series of translations between the languages, is depicted in Figure 3.

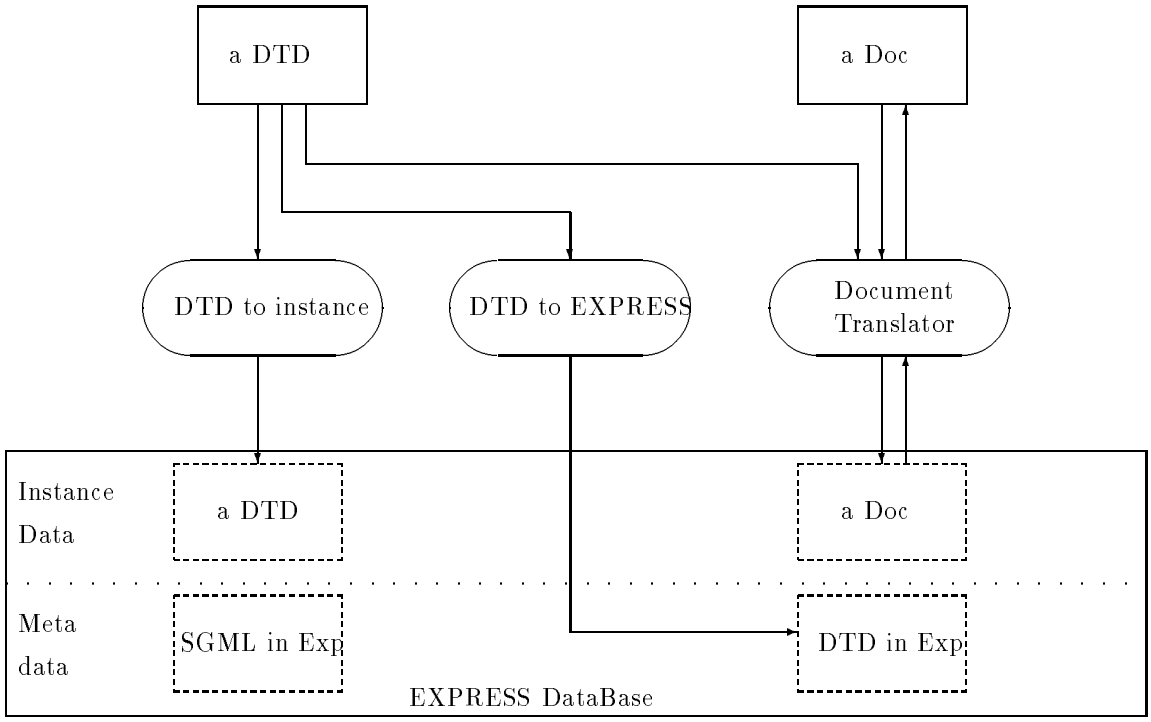

Figure 3: The Architecture of Language Translators.

At the core of the architecture is an EXPRESS database which stores all components of the information model, be they component data, documents, or process models.

In order to store documents in this database and also to store the formats to which they conform, we propose a series of Translators to convert documents and document formats into EXPRESS instance data and meta-data and also extract documents from the database. The translations are as follows:

(1). SGML in EXPRESS. The DTDs themselves are data items which have to be retrieved and reused just like any other. To store DTDs, an EXPRESS schema is predefined in the EXPRESS database which captures the SGML language metadata. DTDs are then translated and stored as instances of this predefined EXPRESS model.

(2). DTD to EXPRESS. An SGML to EXPRESS Translator takes a DTD, and generates an EXPRESS schema which has a similar information structure and content. Instances of this generated schema will thus be documents conforming to the DTD, stored as EXPRESS data.

(3). Document to EXPRESS Translator. A document translator is used to translate documents conforming to the given DTD, which has been analysed by the DTD-to-EXPRESS translator above. Note that this will require the specific DTD as input as well as the document. This will convert the document into an EXPRESS instance of the translated DTD's schema.

(4). EXPRESS to Document Translator. Retrieval of document components from the EXPRESS database requires a reverse translator from EXPRESS to SGML also to be generated from the SGML to EXPRESS mapping so that documents stored in the database can be extracted in a manner corresponding to their SGML definitions.

We give some of the details of these translations in subsequent sections of this document.

4 Representing DTDs in EXPRESS

In this section we discuss the translation of SGML DTDs into instances of the "SGML in EXPRESS" described above. The purpose of this translation is so that DTDs can be handled in the same manner as the rest of the data in the information model.

A generic EXPRESS schema is defined capturing the structure of DTDs. DTDs are then defined as instances of this schema. The EXPRESS schema and the translation are here defined step by step, using the previously defined example DTD.

A DTD is a sequence of SGML declarations, stored in the following entity:

```
ENTITY dtd;
    lines : LIST [0:?] OF sgml_decl;
END_ENTITY ;
```

Thus in our example, we generate the following instance. Note that the instance reference identifiers are automatically generated by the implementation and have do not refer to anything in this paper.

```
report_inst = dtd([#1,#2,#3,#4,#5,...,#17,#18])
```

The three kinds of SGML declaration we are using in our example are represented via an **ABSTRACT** supertype *sgml_decl* which can be one of three subtypes, *entity_decl*, *element_decl*, or *attlist_decl*.

```
ENTITY sgml_decl
      ABSTRACT SUPERTYPE OF
          (ONEOF(entity_decl, element_decl, attlist_decl)) ;
END_ENTITY ;
```

4.1 Elements

Elements become instances of the following EXPRESS entity, with a name, a tag minimisation and a group.

```
ENTITY element_decl;
        element_name : STRING ;
        tag_min : tag_min_pair ;
        content : group ;
END_ENTITY ;
```

The tag minimisation flags are described via the following entity and enumeration type:

```
ENTITY tag_min_pair ;
        begin_min : TagMin ;
        end_min : TagMin ;
END_ENTITY;

TYPE TagMin = ENUMERATION OF (KEEP, OMIT); END_TYPE ;
```

The groups are represented by the following type:

```
TYPE group = SELECT(symbol_group, base_group, seq_group, and_group,
                    or_group, opt_group, closure, ne_closure);
END_TYPE;
```

A *symbol_group* is a reference to another SGML element and is thus just a string, while other base groups are references to the base types of SGML and thus are given by an enumeration of names. The other group types store the complex group declarations.

```
TYPE symbol_group = STRING;
END_TYPE;

TYPE base_group = ENUMERATION OF (PCDATA, CDATA, EMPTY, ...) ;
END_TYPE ;

ENTITY seq_group ;
        s : LIST [0:?] OF group;
END_ENTITY;
...
(* and_group, and or_group as seq_group - omitted for space *)

ENTITY opt_group;
        opt : group;
END_ENTITY;
...
(* closure and ne_closure as opt_group - omitted for space *)
```

Thus in our example, the Title element translates to:

```
#4 = element_decl('Title',#23,PCDATA)
#23 = tag_min_pair(KEEP,OMIT)
```

Whilst the Head element, Body element and paragraph element P translate to the following more complex set of entity instances.

```
#3 = element_decl('Head',#21,#22)
#21 = tag_min_pair(KEEP,KEEP)
#22 = seq_group(['Title','Author'])

#6 = element_decl('Body',#25,#26)
#25 = tag_min_pair(KEEP,KEEP)
#26 = ne_closure('Sections')

#9 = element_decl('P',#30,#31)
#30 = tag_min_pair(KEEP,OMIT)
#31 = or_group([PCDATA,'List','Subsect'])
```

4.2 Attributes

Attributes are recorded using two EXPRESS entities. *attlist_decl* records the element which the attributes refer to and each *sgml_attribute* records the form of an attribute, with a name, a range of values for the attribute and a default.

```
ENTITY attlist_decl;
        element_ref : STRING ;
        attrs : LIST [1:?] OF sgml_attribute ;
END_ENTITY ;

ENTITY sgml_attribute;
        attribute_name : STRING ;
        declared_value : attribute_value ;
        default_value : attribute_default ;
END_ENTITY ;
```

In this restricted subset of SGML, *attribute_value* can either be a keyword, such as CDATA, represented as an enumerated type *enum_attr*, or can be a group attribute, for a list of alternate values, given by the *tok_grp_attr*.

```
TYPE attribute_value = SELECT(enum_attr, tok_grp_attr);
END_TYPE ;

TYPE enum_attr =
        ENUMERATION OF (CDATA_ATT, ... );
END_TYPE ;

ENTITY tok_grp_attr ;
        tok_grp : namegroup ;
END_ENTITY ;

TYPE namegroup = SELECT(base_attr,or_namegroup);
END_TYPE ;

TYPE base_attr = STRING;
```

```
END_TYPE;

ENTITY or_namegroup ;
        o1 : namegroup ;
        o2 : namegroup ;
END_ENTITY;
```

A similar selection occurs with attribute defaults, which in the restricted subset of SGML being presented here, can either be a given string, or a keyword such as REQUIRED.

```
TYPE attribute_default = SELECT(enum_default, default_attr);
END_TYPE ;

ENTITY default_attr;
        def : STRING ;
END_ENTITY;

TYPE enum_default = ENUMERATION OF (REQUIRED, ...);
END_TYPE ;
```

In our example, the following EXPRESS instances represent the attribute declarations.

```
#18 = attlist_decl('Report',[#44,#45,#46])

#44 = sgml_attribute('security',#53,#54)
#53 = tok_grp_attr(#57)
#54 = default_attr('public')
#57 = or_namegroup(['confiden','public'])

#45 = sgml_attribute('status',#55,#56)
#55 = tok_grp_attr(#58)
#56 = default_attr('draft')
#58 = or_namegroup(['draft','final'])

#46 = sgml_attribute('refid',CDATA_ATT,REQUIRED)
```

4.3 Entities

SGML entities[4]are represented by the following EXPRESS entities.

```
ENTITY entity_decl
        ABSTRACT SUPERTYPE OF (ONEOF(external_entity,parameter_entity));
        entity_name : STRING ;
        entity_value : STRING ;
END_ENTITY;
```

[4]This entity allows us to distinguish between *general* entities, which have been discussed and *parameter* entities, which we will not discuss in the current paper.

```
ENTITY external_entity
        SUBTYPE OF (entity_decl) ;
END_ENTITY;

ENTITY parameter_entity
        SUBTYPE OF (entity_decl) ;
END_ENTITY;
```

Thus in our example, the entity RAL represented using the following EXPRESS instance.

```
#1 = external_entity('RAL','Rutherford Appleton Laboratory')
```

5 Generating EXPRESS Schemas

The second translation required takes SGML DTDs and generates EXPRESS schemas. Thus documents which are written to conform to the DTD can be stored as EXPRESS instances of the resulting schema. This section follows and extends the ideas presented in [1].

To do this translation we establish a correspondence between objects in an SGML DTD and elements in an EXPRESS schema definition. There are several ways in which this may be done: we present one which is reasonably straightforward.

This translation uses some general types, which included in every generated schema. These are defined in the next two subsections.

5.1 General Supertypes

Supertypes are defined for the major syntactic categories of SGML.

```
ENTITY Element_SGML
   ABSTRACT SUPERTYPE ;
END_ENTITY ;

ENTITY Entity_SGML
   ABSTRACT SUPERTYPE ;
END_ENTITY ;

ENTITY Attribute_SGML
   ABSTRACT SUPERTYPE ;
END_ENTITY ;
```

The default method for building subtypes of a supertype is to use **ANDOR**. However, in our example this would generate many superfluous subtypes. Hence, we shall assume that the **ONEOF** subtype constructor is adopted as the default here.

5.2 SGML Base Data Types.

SGML provides base types for blocks of text. These are represented using the following EXPRESS types.

A typical block of text in an SGML document, represented as a `PCDATA` block, will contain both standard text and *entity references*. An EXPRESS type which models this is a list of base elements, each of which can be an string or an entity of supertype Entity_SGML.

```
TYPE PCDATA =
    LIST OF SGMLBaseDataType ;
END_TYPE ;

TYPE SGMLBaseDataType =
    SELECT OF (STRING, Entity_SGML) ;
END_TYPE ;
```

The unparsed character data type `CDATA` is interpreted as STRING type, and the *EMPTY* datatype as the enumerated type with the single element *empty*.

```
TYPE CDATA = STRING ;
END_TYPE ;

TYPE EMPTY =
    ENUMERATION OF (empty);
END_TYPE ;
```

An alternative approach to handling SGML entity references might be to replace them in situ with their defined text before translation. However, since the definitions of entity references may be changed without the text being changed, we choose to leave entity references in the translated form.

5.3 Translating Document Type Definitions

An SGML Document Type Definition is translated to an EXPRESS schema. In our example, we generate the following schema skeleton.

```
SCHEMA Report_dtd ;
    ...
END_SCHEMA ;
```

The list of SGML declarations within the DTD are translated in turn to form the entities and types of the generated schema.

There is a problem however, in naming schema. EXPRESS Schemas have a name, however, there is no "general" name for a DTD. DTD's are referred to in documents using the `<!DOCTYPE` declaration by a name which refers to an element in the DTD to which the document must conform. There is also a convention of including a "doctype" SGML *entity* in DTD files, for example an entity declaration such as:

```
<!ENTITY % doctype "Report" -- the Report DTD -->
```

We could adopt a convention that each DTD must have a `doctype` entity which we can use. Alternatively we can use the convention of using the name of the "principle element" which could be the first element to be declared, which is done in the above example.

5.4 Translating SGML Elements

Elements are translated to EXPRESS Entities. These are subtypes of the general entity, *Element_SGML*. Thus the translation an element declaration of the form:

`<!ELEMENT` *element_name tagmin group* `>`

is as follows.

ENTITY *element_name*`_type`
SUBTYPE OF Element_SGML ;
...
END_ENTITY ;

The name of the element is translated to the name of the entity by appending the string "`_type`" and declare the entity to be a subtype of *Element_SGML*. Note that the tag minimisation flags are omitted in this translation process. We assume that the tag minimisation of the document has been handled by the SGML parsing mechanism, so when the document is translated such information has been lost. When documents are retranslated back from EXPRESS to SGML all tags are included.

The form of the groups of the element determine the attributes of the EXPRESS entity. The simplest element group is a base type, such as `#PCDATA`. This translates into a single attribute of type `PCDATA`. This attribute is assigned an arbitrary name, unique within the entity. Thus, in our example, the `Title` (line 6) element translates into the following EXPRESS entity.

ENTITY Title_type
SUBTYPE OF (Element_SGML);
a1 : PCDATA;
END_ENTITY ;

Elements `Author`, `H1`, `H2`, `L1`, `Item` and `A1` are similar.

If the element group has two or more elements in sequence, then each sub-element generates an attribute, the order of the attributes reflecting the order of the sequence. Thus the `Head` element (line 5) is translated into the following entity.

ENTITY Head_type
SUBTYPE OF (Element_SGML);
Title1 : Title_type;
Author2 : Author_type;
END_ENTITY ;

Element `Title` will be translated to entity Title_type so we used that type name and the type for the first attribute: similarly for the `Author` element. Note that we have used the name of the element reference for attribute names, appended with a number to make the attribute unique within the entity. This naming convention makes comprehension easier and is used where possible. Uniqueness of attribute names is important since the same element reference could be used more than once in the same group. For example, a heading with a subtitle may have the element group:

```
(Title, Title, Author)
```

so we would generate the attributes:

Title1 : Title_type;
Title2 : Title_type;
Author3 : Author_type;

The definition of `Body` (line 8) specifies a non-empty recurrence of `Section`s. The translation uses the EXPRESS list type, with minimum bound of 1, as the type of a single attribute, again with a dummy name, in the entity generated for Body.

ENTITY Body_type
    SUBTYPE OF (Element_SGML);
    a1 : **LIST** [1:?] **OF** Section_type ;
END_ENTITY ;

Similar translations apply for the possibly empty recurrence. This translates to the EXPRESS list type, with minimum bound of 0. Thus `Section` (line 9) translates to:

ENTITY Section_type
    SUBTYPE OF (Element_SGML);
    H11 : H1_type;
    a2 : **LIST** [0:?] **OF** P_type ;
END_ENTITY ;

The `Report` (line 4) element translates to an entity including an optional valued attribute representing optional element reference, as follows.

ENTITY Report_type
    SUBTYPE OF (Element_SGML);
    Head1 : Head_type;
    Body2 : Body_type;
    a3 : **OPTIONAL** Appendix_type;
    ...
END_ENTITY ;

The element group which specifies a choice of elements translates to an EXPRESS **SELECT** type. However, we cannot place such a type in line for an attribute type, and we have to generate an auxiliary SELECT type, with a unique dummy name. Thus the element `P` translates to the following:

```
ENTITY P_type
        SUBTYPE OF (Element_SGML);
        a1 : or1_type;
END_ENTITY ;

TYPE or1_type =
     SELECT OF (PCDATA,List_type,Subsect_type);
END_TYPE ;
```

This method of using auxiliary types could, for consistency, be used for all groups, but we chose the approach presented above for clarity. However, there are occasions such as nested groups, when it may be necessary to generate auxiliary types for other groups.

Perhaps the most awkward of all the element structuring operators in SGML to translate satisfactorily into EXPRESS is the "`&`" operator. This specifies that the elements of the group can occur in any order. We translate the group in a similar way to a sequence of elements. Thus the element `List` (partially) translates into:

```
ENTITY List_type
        SUBTYPE OF (Element_SGML);
        L11 : L1_type;
        a2 : LIST [1:?] OF Item_type ;
        ...
END_ENTITY ;
```

However, the attributes of an EXPRESS entity occur in a fixed order. Thus if we translate the elements in order, we lose information; when the document is regenerated from the EXPRESS schema, the correct ordering of the elements may be lost. Thus we need to record the order in which sub-elements occur. We do this by adding another attribute to the entity, which is a list of numbers recording the order. Thus we add to the *List_type* entity:

```
and_ordering_3 : LIST [2:2] OF UNIQUE INTEGER ;
WHERE and_ordering_condition3:
        is_permutation(and_ordering_3, [1,2]) ;
```

This attribute *and_ordering_3* is a non-repeating list of length two, which is the number of elements in the group. We also add a **WHERE** clause using an auxiliary function *is_permutation* to assert that *and_ordering_3* is a permutation of the list $[1, 2]$.

5.5 Translating SGML Attributes

Each SGML attribute translates to an entity which is a subtype of the abstract supertype *Attribute_SGML*. An auxiliary enumeration type is used to represent any alternative values of the attribute. Thus the three attributes in the example are represented as follows:

```
ENTITY security_type
        SUBTYPE OF (Attribute_SGML);
        a1 : attr9_type;
END_ENTITY ;

TYPE attr9 type =
     ENUMERATION OF (confiden,public);
END_TYPE ;

ENTITY status_type
        SUBTYPE OF (Attribute_SGML);
        a1 : attr10_type;
END_ENTITY ;

TYPE attr10_ type =
     ENUMERATION OF (draft,final);
END_TYPE ;

ENTITY refid_type
        SUBTYPE OF (Attribute_SGML);
        a1 : CDATA;
END_ENTITY ;
```

Each SGML attribute is associated with an element. In order to make the association explicit in the EXPRESS model, attributes are added to the translation of the relevant SGML element. In our example, the SGML attributes generate three further EXPRESS attributes of the *Report_type* entity as follows.

```
ENTITY Report_type
        SUBTYPE OF (Element_SGML);
        Head1 : Head_type;
        Body2 : Body_type;
        a3 : OPTIONAL Appendix_type;
        security4 : security_type;
        status5 : status_type;
        refid6 : refid_type;
END_ENTITY ;
```

We do not cover the translation of attribute defaults in this paper.

5.6 Translating SGML Entities

SGML entities are translated into EXPRESS entities which are subtypes of the abstract supertype *Entity_SGML*. However, the value of a SGML entity is set to a constant in the DTD. We thus represent the value of the SGML entity by defining a **DERIVED** attribute, with a dummy name, and declaring that the value of that attribute is set to a constant value. Thus the entity `RAL` in the example (line 2) is translated to the following.

ENTITY RAL
    SUBTYPE OF (Entity_SGML);
    DERIVED d: **STRING** := 'Rutherford Appleton Laboratory'
END_ENTITY ;

This may not be the best way to translate entities. For example, it may be more fruitful to translate them as constants.

6 Generating Documents from EXPRESS

Having generated the above EXPRESS model for any given DTD, it is straightforward to translate document instances and store them in the database. We then have the task of extracting documents from the database and converting them into a form that can be processed by a document generation system. This is also straightforward and will not be discussed further.

Extraction is more difficult when we wish to use data from the EXPRESS model, but not originating from a document. For example, reports documenting technical data on the components of a control system will contain data on those components extracted from an EXPRESS representation of the component, rather than from the representation of the report.

In order to construct documents which can refer to other data represented in EXPRESS, we have developed the concept of "*prototype documents*". These are documents with 'holes' embedded in their text which will be filled in by data extracted from EXPRESS entities when the document is transformed into its SGML form. These holes specify which EXPRESS entity is referred to and which attribute of the entity is to be displayed: thus the holes are more properly described as "*EXPRESS references*". As documents are being represented in SGML, these EXPRESS references are given an SGML syntax as follows which can be added to each DTD.

```
<!ELEMENT  ExpRef  - O    EMPTY           >
<!ATTLIST  ExpRef  ent    CDATA #REQUIRED
                   expid  CDATA #REQUIRED
                   att    CDATA #REQUIRED  >
```

We allow any text in the document to contain EXPRESS references in addition to standard SGML text.

```
<!ELEMENT  Text  O O  (#PCDATA | ExpRef)*  >
```

Thus at any point in the text of the document we can include an EXPRESS reference, which is just a start tag with no body (as the `EMPTY` group declaration is used), and no end tag. Each EXPRESS reference has three attributes which are used to refer into the EXPRESS instance. The first *ent* is an entity name giving a type of entity; the second

expid is an internal object identifier so that the document can refer to a specific instance in the data base[5], and the third *att* is an EXPRESS attribute name. All three are required attributes in the SGML tag.

Thus for example, in a typical control system, we might have delays, which may be represented by the following partial EXPRESS entity.

ENTITY Delay ;
    ref_num : **INTEGER** ;
    timeout : **INTEGER** ;
    ...
END_ENTITY ;

And there maybe several instances of this delay used in the system:

delay1 = Delay(1, 30) ;
delay2 = Delay(2, 10) ;
...

The technical description of this delay might include the following sentence in a prototype document:

```
Delay number <ExpRef ent=Delay expid=delay2 att=ref_num> has a time
out of <ExpRef ent=Delay expid=delay2 att=timeout> seconds.
```

On generation of the final document, these EXPRESS references will be expanded out to generate the desired text:

```
Delay number 2 has a time out of 10 seconds.
```

The *expid* attribute itself could be a parameter to the document. The document could then be parameterised over any EXPRESS database so that we can have reusable documents.

This approach is the first step on integrating EXPRESS data into reusable documents, and still under development. These problems are discussed further in a draft paper [5].

7 Conclusions

In this paper we have presented an architecture for integrating document modelling in SGML with data modelling and storage in EXPRESS. The central mechanism of this integration is a series of translation functions between the languages which allow the representation of one form of data in another. A sketch of these translations has been given in respect of an example document.

The translation functions sketched in this paper cover only a subset of the full SGML language. There are further aspects of the language which have yet to consider. These include:

[5] We assume that there is a mechanism to make such identifiers available

- **Attribute Defaults.** These are only partially covered in the representation of DTDs and not at all in the generation of schemas from EXPRESS. More consideration is needed to cover defaults comprehensively.

- **Entities.** We only give the most simple use of SGML entities. There are more sophisticated uses of entities which we do not consider here.

- **Inclusions and Exclusions.** SGML provides a mechanism whereby elements can be freely included within other elements (*inclusions*), or excluded from elements in given circumstances (*exclusions*). The above translations do not cover this aspect of the language.

Also, as mentioned above, further work is required in the area of extracting data from an EXPRESS model into documents.

A prototype implementation of the translation functions described above has been begun in the functional language Standard ML. Work is continuing to extend and improve this implementation

The TORUS project is primarily interested in *reuse* of documents and data. This involves combination, parameterisation and inheritance of documents and sub-documents. This is poorly covered by the current SGML standard and will require further investigation.

Acknowledgements

We would like to thank our partners on the TORUS project, and also our colleagues at RAL for help and advise on this paper, particularly Brian Ritchie for his work on extracting documents from EXPRESS data.

References

[1] Wey-tyng Chang *"Comparison of Two Information Modeling Languages: SGML and EXPRESS."* EXPRESS User Group Conference, 1994.

[2] *"Information Processing - Text and Office Systems - Standard Generalised Markup Language (SGML)"*, ISO Standard 8807, 1988.

[3] *"Product Data Representation and Exchange - Part 11: The EXPRESS Language Reference manual"*, ISO Standard 10303-11 (TC 184/SC4), 1994.

[4] B.M.Matthews and J.C.Bicarregui, *"Process Modelling in Control Systems Design"* to appear EUG'95, Express User Group Conference, Grenoble, France, October 1995.

[5] B. Ritchie *"Issues in the Reuse of Documentation through Prototypes."* ESPRIT 8080 TORUS Project draft document TORUS/RAL/20/1, April 1995.

[6] Wenzel, B., et. al., "*Interoperability between STEP and SGML*" Swedish Association for CALS White Paper, appeared in Engineering Data Newsletter April and June 1995.

Juan Bicarregui.
Juan Bicarregui has a BSc and MSc in Mathematics from Imperial College and Queen Mary College, London and a PhD in Computer Science from the University of Manchester under the supervision of Prof. C. B. Jones.

His research interests are in the application of formal methods and formal notations to software engineering. This work has included the development of a specification and proof assistant, Mural, and case studies in the use of VDM and B. He recently co-authored a practitioners guide to proof in VDM. His current interests are in the application and integration of formal notations including EXPRESS, SGML.

Brian Matthews.
Brian Matthews has a BSc in Mathematics and a MSc in the Foundations of Advanced Information Technology. He is currently completing a PhD in equational reasoning at the University of Glasgow. He has been working in the Software Engineering Group of the Rutherford Appleton Laboratory since 1986. He is interested in advanced techniques for software engineering, especially in the areas of formal methods and tools, using both the B notation and equational reasoning, and also in the integration of methods, notations and standards.

DTD⟦ _ ⟧ : Document Type Definition ⟶ EXPRESS Schema

DTD⟦ *dtd* ⟧ = **SCHEMA** "dtd-name" ;
D⟦ *dtd* ⟧ *built-ins*
END_SCHEMA ;

D⟦ _ ⟧ _ : SGML Declarations × EXPRESS Declarations ⟶ EXPRESS Declarations

D⟦ *dec decs* ⟧ *typs* = D⟦ *decs* ⟧ (L⟦ *dec* ⟧ typs)

L⟦ _ ⟧ _ : SGML Element Declaration × EXPRESS Declarations ⟶ EXPRESS Declarations

L⟦ `<!ELEMENT` element_name tagmin group `>` ⟧ *typs* =
ENTITY *element_name*`_type`
SUBTYPE OF Element_SGML ;
as
END_ENTITY ;
:: *typs′*
where (*as*,*typs′*) = G⟦ group ⟧ *typs*

G⟦ _ ⟧ _ : SGML Element Group × EXPRESS Declarations ⟶ EXPRESS Attributes × EXPRESS Declarations

$$
\begin{array}{lcl}
\mathrm{G}[\![\ elem\_name\ ]\!]\ typs & = & ([elem\_name : elem\_name\_\texttt{type}\ ;],typs) \\
\mathrm{G}[\![\ base\_type\ ]\!]\ typs & = & ([a\_name : base\_type\ ;],typs) \\
 & & \text{where } a\_name = \text{NewName ()} \\
\mathrm{G}[\![\ g\_1\ ,\ g\_2\ ,\ \ldots,\ g\_n\ ]\!]\ typs & = & (a::as,\ typs'') \\
 & & \text{where } ([a],\ typs') = \mathrm{G}[\![\ g\_1\ ]\!]\ typs \\
 & & \qquad (as,\ typs'') = \mathrm{G}[\![\ g\_2\ ,\ \ldots,\ g\_n\ ]\!]\ typs' \\
\mathrm{G}[\![\ g\_1 \mid \ldots \mid g\_n\ ]\!]\ typs & = & ([a\_name : group\_\texttt{type}],typs) \\
 & & \text{where } a\_name = \text{NewName ()} \\
 & & \qquad (group\_\texttt{type},typs') = \mathrm{T}[\![\ g\_1 \mid \ldots \mid g\_n\ ]\!]\ typs \\
\mathrm{G}[\![\ g\_1\ \&\ \ldots\&\ g\_n\ ]\!]\ typs & = & ([a\_name : group\_\texttt{type}],typs) \\
 & & \text{where } a\_name = \text{NewName ()} \\
 & & \qquad (group\_\texttt{type},typs') = \mathrm{T}[\![\ g\_1\ \&\ \ldots\&\ g\_n\ ]\!]\ typs \\
\mathrm{G}[\![\ g?\ ]\!]\ typs & = & ([a\_name : \text{OPTIONAL } group\_\texttt{type}],typs') \\
 & & \text{where } a\_name = \text{NewName ()} \\
 & & \qquad (group\_\texttt{type},typs') = \mathrm{T}[\![\ g\ ]\!]\ typs \\
\mathrm{G}[\![\ g*\ ]\!]\ typs & = & ([a\_name : group\_\texttt{type}],typs') \\
 & & \text{where } a\_name = \text{NewName ()} \\
 & & \qquad (group\_\texttt{type},typs') = \mathrm{T}[\![\ g*\ ]\!]\ typs \\
\mathrm{G}[\![\ g+\ ]\!]\ typs & = & ([a\_name : group\_\texttt{type}],typs') \\
 & & \text{where } a\_name = \text{NewName ()} \\
 & & \qquad (group\_\texttt{type},typs') = \mathrm{T}[\![\ g+\ ]\!]\ typs
\end{array}
$$

$\mathrm{T}[\![\ \_\ ]\!]\ \_$: SGML Element Group $\times$ EXPRESS Declarations $\longrightarrow$ EXPRESS Type Name $\times$ EXPRESS Declarations

T⟦ *elem_name* ⟧ *typs* = (*elem_name*_`type`,*typs*)

T⟦ *base_type* ⟧ *typs* = (*base_type*,*typs*)

T⟦ *g_1* , ..., *g_n* ⟧ *typs* = (*t_name*, *typs''*)
where *t_name* = NewName ()
([*at_1* ... *at_n*],*typs'*) = fold (G⟦ _ ⟧ _) typs [*g_1* ... *g_n*]
typs'' = *typs'* ::
ENTITY *t_name* ;
[*at_1* ... *at_n*]
END_ENTITY ;

T⟦ *g_1* | ...| *g_n* ⟧ *typs* = (*t_name typs''*)
where *t_name* = NewName ()
([*g_1_name* ... *g_n_name*], *typs'*) = fold (T⟦ _ ⟧ _) typs [*g_1*
typs'' = *typs'* ::
TYPE *t_name* =
SELECT [*g_1_name* ... *g_n_name*] ;
END_TYPE ;

T⟦ *g_1* & ...& *g_n* ⟧ *typs* = (*t_name typs''*)
where *t_name* = NewName ()
s_name = NewName ()
([*g_1_name* ... *g_n_name*], *typs'*) = fold (T⟦ _ ⟧ _) typs [*g_1*
n = # [*g_1_name* ... *g_n_name*]
typs'' = *typs'* ::
TYPE *t_name* =
LIST [*n*:*n*] **OF** *s_name* ;
WHERE
SIZEOF(**QUERY**(*x* <∗**SELF**|**TYPEOF**(*x*) = *g_*
= occurences(*g_1_name*,[*g_1_name* ... *g_n*
...
SIZEOF(**QUERY**(*x* <∗**SELF**|**TYPEOF**(*x*) = *g_*
= occurences(*g_n_name*,[*g_1_name* ... *g_*
END_TYPE ; ::
TYPE *s_name* =
SELECT [*g_1_name* ... *g_n_name*] ;
END_TYPE ; ::

T⟦ *g?* ⟧ *typs* = (*t_name typs''*)
where *t_name* = NewName ()
(*g_name*,*typs'*) = T⟦ *g* ⟧ *typs*
typs'' = *typs'* ::
TYPE *t_name* =
LIST [0:1] **OF** *g_name* ;
END_TYPE ;

T⟦ *g∗* ⟧ *typs* = (*t_name typs''*)
where *t_name* = NewName ()
(*g_name*,*typs'*) = T⟦ *g* ⟧ *typs*
typs'' = *typs'* ::
TYPE *t_name* =
LIST [0:?] **OF** *g_name* ;
END_TYPE ;

T⟦ *g+* ⟧ *typs* = (*t_name typs''*)
where *t_name* = NewName ()
(*g_name*,*typs'*) = T⟦ *g* ⟧ *typs*
typs'' = *typs'* ::
TYPE *t_name* =

A⟦ _ ⟧ _ : SGML Attlist × EXPRESS Declarations ⟶ EXPRESS Declarations

A⟦ <!ATTLIST *element_name attdeclist* > ⟧ *typs* = *typs″*
where *e* = find_element *typs element_name*
([*a_1* ... *a_n*], *typs′*) = fold B⟦ _ ⟧ _ *typs attdeclist*
e′ = add_attributes *e* [*a_1* ... *a_n*]
typs″ = change_element *typs′ e e′*

B⟦ _ ⟧ _ : SGML AttDec × EXPRESS Declarations ⟶ EXPRESS Attribute × EXPRESS Declarations

B⟦ *attname attvals attdef* ⟧ *typs* = (*attname*:*attname*_type; , *typs″*)
where ([*at_1* ... *at_n*],*typs′* = V⟦ *attvals* ⟧ *typs*
typs″ = *typs′* ::
ENTITY *attname*_type ;
SUBTYPE OF Attribute_SGML ;
[*at_1* ... *at_n*]
END_ENTITY ;

V⟦ _ ⟧ _ : SGML Attribute Value Group × EXPRESS Declarations ⟶ EXPRESS Attributes × EXPRESS Declarations

This is similar to G⟦ _ ⟧ _ and is omitted here for the time being.

N⟦ _ ⟧ _ : SGML Entity Dec × EXPRESS Declarations ⟶ EXPRESS Declarations

N⟦ <!ENTITY *entity_name entity_value* > ⟧ *typs* = *typs′*
where *typs′* = *typs′* ::
ENTITY *entity_name* ;
SUBTYPE OF Entity_SGML ;
DERIVED *d*:**STRING** := '*entity_value*
END_ENTITY ;