A Structural EXPRESS Editor

Thomas R. Kramer

Katherine C. Morris

David A. Sauder

NISTIR 4903
August 17, 1992

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.

Copyright

Acknowledgment

Partial funding for the work described in this document was provided to Catholic University by the National Institute of Standards and Technology under cooperative agreement Number 70NANB2H1213.

1 Introduction

The Structural EXPRESS Editor (SEXE) is a prototype editing system for writing schemas in the EXPRESS information modeling language. It runs on a computer running the X Window System and has a friendly user interface. It is called a structural editor because the user interacts with the system to write and edit the logical structure of EXPRESS statements, not the actual text of the statements. The SEXE editing system has two main parts:

- an Interactive Editor, built by an editor building tool called the Data Probe Editor Builder (DPE Builder), and
- a Translation Module which reads the output of the Interactive Editor and writes EXPRESS.

1.1 SEXE is a Prototype

The SEXE editing system is a prototype; it implements only a portion of EXPRESS. The DPE Builder is not yet complete itself, so some functionality that appears to be in the SEXE editing system is not, in fact, present. The Interactive Editor produced by the DPE Builder has not been modified to adapt it further to EXPRESS. To make the SEXE editing system a useful tool for producing EXPRESS schemas would require a great deal of additional work. Building and using the prototype, however, has revealed a lot about alternate representations of EXPRESS and about desirable features of EXPRESS editors.

1.2 About this Paper

Section 2 of this paper gives the background for this paper: what EXPRESS and STEP are, what work NIST is doing related to STEP, and what the motivation is for building an EXPRESS editor. Section 3 describes the DPE Builder. Section 4 describes the two parts of the SEXE editing system: the editor built by the DPE Builder and the Translation Module. That section also gives a brief example of using the SEXE editing system. Section 5 presents alternative representations of EXPRESS. Section 6 briefly discusses some issues in information modeling highlighted in the course of building the SEXE editing system. Section 7, the conclusion, assesses the SEXE editing system, discusses EXPRESS editors generally, and presents ideas for future related work.

This paper is not a users manual for the SEXE editing system.

The paper is intended to be of interest to persons studying information modeling, particularly the EXPRESS language, and to persons developing STEP tools, such as EXPRESS editors.

2 Background

This section discusses EXPRESS and STEP, describes work at NIST related to STEP, and gives the motivation for building an EXPRESS editor.

2.1 EXPRESS and STEP

The next generation of product information standards, commonly referred to as the Standard for the Exchange of Product Model Data (STEP), is being developed within the International Organization for Standardization (ISO). The STEP effort has included the development of basic methods for information representation. The STEP standard is divided into documents called Parts. The two STEP Parts of interest here are Part 11 and Part 21. There are many versions of both documents. The versions used in the SEXE editing system are cited below. For an overview of the STEP standards, refer to Part 1 [ISO].

Part 11, entitled “EXPRESS Language Reference Manual,” is the definition of the official STEP information modeling language, EXPRESS. We are using the April 1991 version [Spiby91a], also known as the “N14”version.

Part 21, generally called the “Exchange File Rules”, gives rules for how to write files containing instances of information entities defined in EXPRESS. Methods other than exchange files are being developed to support data sharing - in particular, the STEP Data Access Interface (SDAI) - but they will not be discussed here. The SEXE editing system conforms to the 1988 version of the Exchange File Rules given in [Altemueller88a] and [Altemueller88b], except that forward references are permitted by the SEXE editing system, since the capability to write EXPRESS would be incomplete without them, and they are allowed in more recent versions of the Exchange File Rules. Several more recent versions of the Exchange File Rules have appeared, the most recent of which is [Van Maanen].

The EXPRESS language is being used within STEP to build information models called “schemas” for various purposes: to define the physical shape of products or the manufacturing process plans for products, for example. The relationship between EXPRESS schemas and exchange files is defined so that only one EXPRESS schema is needed for information in a given domain, but a separate exchange file is required for each product to be described using the schema. For example, a schema for topology and geometry may be used to specify how to describe shape. Then, for each object to be modeled, there will be an exchange file containing the instances of information entities from the schema which describe the shape of that particular object.

EXPRESS is readily human-readable. Anyone familiar with EXPRESS can pick up any EXPRESS file and read it. STEP exchange files, which will be called “exchange files” henceforth, are generally difficult to read. Making sense of an exchange file requires a knowledge of both the rules for constructing exchange files and the EXPRESS schema underlying the file. Even then, since the attribute names in exchange file instances are not given (they must be inferred from the position of the attribute value within the instance) and the attribute values may be references to other instances, making sense of

the data which defines an instance is very tedious. Generally, the only easy-to-understand aspects of an instance in an exchange file are the identifying number and the name of the type of entity being instantiated.

Although EXPRESS and the exchange file format were defined for STEP, they and the methods for using them are powerful tools that may be put to good use outside of STEP by anyone doing information modeling.

2.2 Work at NIST Related to STEP

In the Factory Automation Systems Division at the National Institute of Standards and Technology (NIST), a National PDES Testbed has been established (PDES = Product Data Exchange Using STEP). Funding for the Testbed Project has been provided by the Department of Defense's Computer-Aided Acquisition and Logistic Support (CALS) Office. One of the missions of the Testbed has been to develop tools for handling EXPRESS schemas and exchange files. A suite of tools has been developed, and more are under development. The Data Probe Editor Builder (DPE Builder), briefly described in [Morris91], is one of these under development. The DPE builder takes an EXPRESS schema as input and builds an interactive editor for editing exchange files that correspond to the schema.

The NIST Robot Systems Division is interested in applying STEP standards and tools to world modeling for robots and machine tools and to real-time control systems. One of the authors of this paper has previously built EXPRESS schemas for: (i) controller hierarchies, (ii) process plans for 3-axis machining, (iii) process plans for robotic deburring, and (iv) material removal volumes [Kramer]. He also has built editors using the DPE Builder for generating exchange files corresponding to two of the schemas.

2.3 Motivation for Building an EXPRESS Editor

The thinking that provided the motivation for building the SEXE editing system went as follows:

> We and others have been using text editors to write EXPRESS schemas. Text editors have the drawbacks of (1) requiring the user to learn all the details of the format of EXPRESS, (2) not providing the user with help in generating correct EXPRESS statements, and (3) not providing any method of checking a schema when it is finished.
>
> The third drawback is easily addressed by running the finished schema through an EXPRESS parser, such as the NIST Fed-X parser [Clark92], to check the grammar and syntax of the schema.
>
> It is clearly desirable to have an editor which avoids drawbacks 1 and 2, as well. A structure editor does not require the user to know the syntax of the language being edited. Thus, any editor built by the DPE Builder does not have drawback 1, since every such editor is a structure editor. However, the files generated by such editors are exchange files, not EXPRESS files. A translation module will be needed that writes EXPRESS from the files output by an EXPRESS structure editor. The translation module will generate correct EXPRESS, avoiding

drawback 2 of text editors. Thus, using a structure editor built by the DPE Builder together with a translation module can make it possible to avoid drawbacks 1 and 2. To use the DPE builder in this manner, we must first have an EXPRESS schema for EXPRESS to feed into the DPE Builder.

The SEXE editing system can also be a mechanism for experimenting with the feel of a structural editor for EXPRESS in order to develop ideas for what a hand-crafted structural EXPRESS editor might include.

Finally, building an EXPRESS editor using an exchange file editor builder highlights some intriguing issues in information modeling. The idea of using an information modeling language to describe itself is not new, but it is rarely necessary to do that. The SEXE Interactive Editor, however, cannot be built without writing an EXPRESS schema for EXPRESS.

The opportunity to create alternate representations of EXPRESS was also appealing. Once the notion of using the DPE Builder to make a STEP editor had been conceived, it was clear that it would be possible to build an exchange file representation of EXPRESS, and the temptation to do that was irresistible.

Since this editor was envisioned as a prototype only, and the idea of building the editor stemmed from the fact that the major component (the DPE Builder) of a feasible architecture already existed, no further consideration was given to the system architecture.

2.4 Other EXPRESS Editors

We are aware of several other efforts towards making it easier to write EXPRESS. We do not have experience in using these systems.

- An EXPRESS mode for the Epoch text editor (a modified GNU Emacs) was developed by Brion D. Sarachan and Michael A. Kinstrey at G. E. for the DARPA Initiative in Concurrent Engineering. There is an anonymously authored users guide but we know of no other documentation.
- An EXPRESS-G editor, whose development was funded by NIST, was built by Villarica at the University of Syracuse. We know of no documentation.
- An EXPRESS editor which runs on a PC under Microsoft Windows has been built by Mike Yinger of Yinger and Associates. This editor combines text editing and structure editing. It contains a parser to check the correctness of complete schemas or portions of schemas, such as entity declarations. It also includes several methods of tracing the associations of EXPRESS constructs with one another. Reference [Yinger] is 1-page description.
- An EXPRESS editor named “Xpresso” that runs in an X Window environment was reported in [Jasnoch].
- An EXPRESS editor named “Exploit” was reported in [Rösch].
- “DECexpress Graphical Editor”, a diagramming tool for the creation of EXPRESS-G models and the automatic generation of EXPRESS schemas has been announced by Digital Equipment.

Among other types of tools for EXPRESS, EXPRESS Editors are discussed in [Wilson].

3 Data Probe Editor Builder

The Data Probe Editor Builder[1] is a utility that reads an EXPRESS schema and builds an editor that runs in an X Window environment for creating or editing exchange files corresponding to the schema. The EXPRESS schema is hard-coded into source code for the editor by the DPE Builder. The editor built by the DPE Builder for the SEXE editing system is called the Interactive Editor in this paper.

It is not the purpose of this paper to discuss the DPE Builder in detail. There are plans to do that separately. Here we present a brief description only.

The DPE Builder is built in C++ and uses the Fed-X-Plus C++ class generator [Morris90] and the Interviews [Linton] user interface toolkit. It is being developed further by two of the authors (Morris and Sauder).

3.1 Windows of Editors Built by Data Probe Editor Builder

The appearance and behavior of all editors built by the DPE Builder is the same. Figure 1 shows the screen of a workstation running the SEXE Interactive Editor. The editor uses four persistent windows, all of which may be manipulated in the typical ways for windows in the X environment (move, open, close, hide, expose, resize, etc.):

1. the window from which the editor was started (upper left of Figure 1),
2. a Data Probe management window (upper right of Figure 1),
3. an Entity Instance List window (lower left of Figure 1), and
4. an Entity Type List window (lower right of Figure 1).

The last three of these are created by the editor when it starts up. In addition, any number of temporary STEP Entity Editing (SEE) windows may be created and destroyed as the editor is being used. A single SEE window is shown in the middle of Figure 1, partly obscuring the Entity Instance List window.

In the figure, the user is editing a schema_block in the SEE window. The “schema_id” attribute is highlighted, indicating that the user may enter a value for the schema_id.

3.1.1 Starting Window

The window from which the editor is started is used only to display messages, such as “auto-saving” and error reporting, which inform the user about what is happening in the editor.

---

1. The DPE Builder is not yet complete, but it is sufficiently complete to build useful editors. The resulting editors do not have all the functionality they will have when the DPE Builder is complete.

3.1.2 Data Probe Management Window

The Data Probe management window provides the user with system functions such as saving or appending files, clearing the Entity Instance List, and quitting the editor. There is also a subwindow for brief messages to the user. In the figure, that window says “Create Instance”.

3.1.3 Entity Type List Window

The Entity Type List window contains a scrollable list of all the entities defined in the EXPRESS schema used to create the editor. Instances of these entities are what an exchange file is made from. In Figure 1, the first few entities in this window are aggregation_types, base_type, and block. The schema_block entity is highlighted in this window because the user has selected it. Below the list of entities there are some command buttons.

This window is used to select the type of entity to create when the user decides to create an instance of an entity. The user simply selects the name of the entity type to be created with the mouse and then selects the “create” command button. This causes a SEE window to pop up for a new instance of that entity type.

3.1.4 Entity Instance List Window

The Entity Instance List window has two main parts. The top part has the appearance of the “DATA” section of an exchange file. Entity instances are listed one instance per line. An indication of the editing status of an instance may be printed just before the instance. In Figure 1 this window shows all the instances required for the simple geometry schema presented in section 4.3.

The bottom part of the window contains the controls for a searching utility that may be used to find text strings in the top part. The bottom part also contains a set of command buttons to do things to instances: delete, modify, view, save, etc. To use the command buttons, an instance is selected with the mouse, and then the desired command button is selected. If either viewing or modifying is selected, a SEE window pops up.

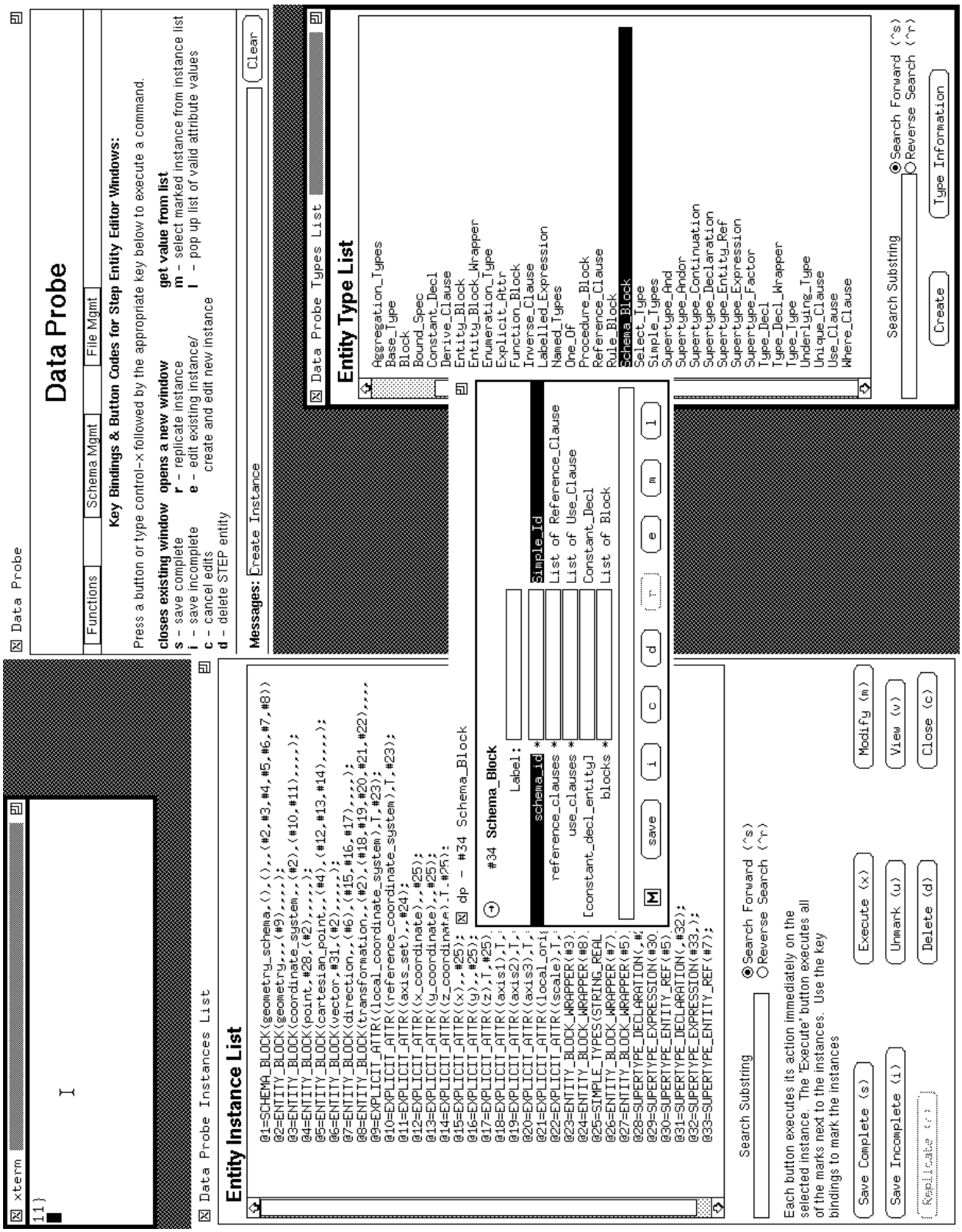

Figure 1. Editor Windows

3.1.5 SEE Windows

SEE (STEP Entity Editing) windows are created and appear when a command for creating or editing an entity instance is given. They disappear (unless “pinned”) when a command is given to save the instance, delete the instance, or close the window. A single SEE window is shown in the middle of Figure 1.

The SEE window for an instance lists the entity type and the identification number of the instance at the top of the window. Identification numbers are assigned to entity instances in numerical order. The remainder of the window consists mostly of lines for the attributes of the entity, one attribute per line. Each line has three sections: the name of the attribute, a space for the user to enter the value of the attribute, and a description of the required type of the attribute. When an entity instance is first created, the middle section of each line is blank. When an entity instance is edited, blank values may be filled in or existing values changed.

Each SEE window has a “save” button at the bottom. When this button is selected, the window disappears, and the contents of the window are transcribed to the Entity Instance List.

The SEE window helps prevent errors by refusing attempts to enter invalid data for attribute values and by checking data types again when an instance is transcribed from the SEE window to the Entity Instance List. Invalid data is not transcribed.

If the value of an attribute is to be an entity instance, an instance of the appropriate type may be created by selecting the attribute line with the mouse, and selecting the E (for edit) button in the SEE window. A new SEE window pops up of the appropriate type.

3.2 Typical Use of the Editor

Typical uses of an editor built by the DPE Builder include reading in and editing existing exchange files or creating new files from scratch. To create a new exchange file, a user starts the editor. Three new windows appear, as described above. The user selects an entity name from the Entity Type List window and selects the create command. A SEE window pops up for an instance of that type of entity with identification number 1. The user selects the attributes in order and fills in the values of each attribute, using the keyboard. When as many values have been filled in as desired, the user selects the save command button, causing the SEE window to disappear and the values that were entered in the SEE window to appear as the values of that entity instance in the Entity Instance List window.

Then the user selects another entity name from the Entity Type List window, selects the create command button, and another SEE window appears, with identification number 2. This window is filled in, and so on.

Often it will be necessary to go back to previously defined instances and modify them to insert values which are references to instances that did not exist when the instance being modified was first defined.

When the user is done, the contents of the Entity Instance List window are saved to an exchange file, and the user exits from the editor.

3.3 Editor Output

The editor will save the contents of the Entity Instance List to a file in either of two formats: normal exchange file format, or working file format (which is like exchange file format with a few additional markings to indicate the state of instances in the editing session). The exchange file the editor writes is nearly identical to the contents of the Entity Instance List window when the file is created. Figure 3 is the exchange file created by an editor for the instances shown in the Entity Instance List window of Figure 1.

4 The SEXE Editing System

The SEXE editing system has two major components: the Interactive Editor and the Translation Module. Figure 2 shows components and data flow for the SEXE editing system.

The vertical box on Figure 2 labelled “Build Interactive Editor” shows how the SEXE Interactive Editor was built. This was done by the system builders. The EXPRESS schema for the EXPRESS language defined by the system builders was fed into Fed-X-Plus, which produced corresponding C++ classes. These were fed into the Data Probe Editor Builder, which built the SEXE Interactive Editor.

The horizontal box on Figure 2 labelled “Use SEXE Editing System” shows how data flows through the components of the SEXE editing system when it is being used to write EXPRESS schemas. The user interacts with the SEXE Interactive Editor to produce an exchange file for an EXPRESS schema for whatever domain the user chooses. The file is then fed into the SEXE translator module, which outputs the schema in correct EXPRESS syntax. This may be done as many times as the user wishes.

4.1 Interactive Editor

The Interactive Editor is the EXPRESS structure editor produced by the DPE Builder using the EXPRESS schema for EXPRESS shown in Appendix C as input.

When a user is using the Interactive Editor, he or she interacts with the list of Entity Instances through the SEE windows described above. These windows show the logical structure of the object being edited, not any form of output text. The logical structure is a list of attributes of the object for which the user provides values, many of which may be references to other objects. This logical view of what is being edited applies equally well to EXPRESS objects and exchange file objects, so the SEE windows are quite well suited to editing EXPRESS.

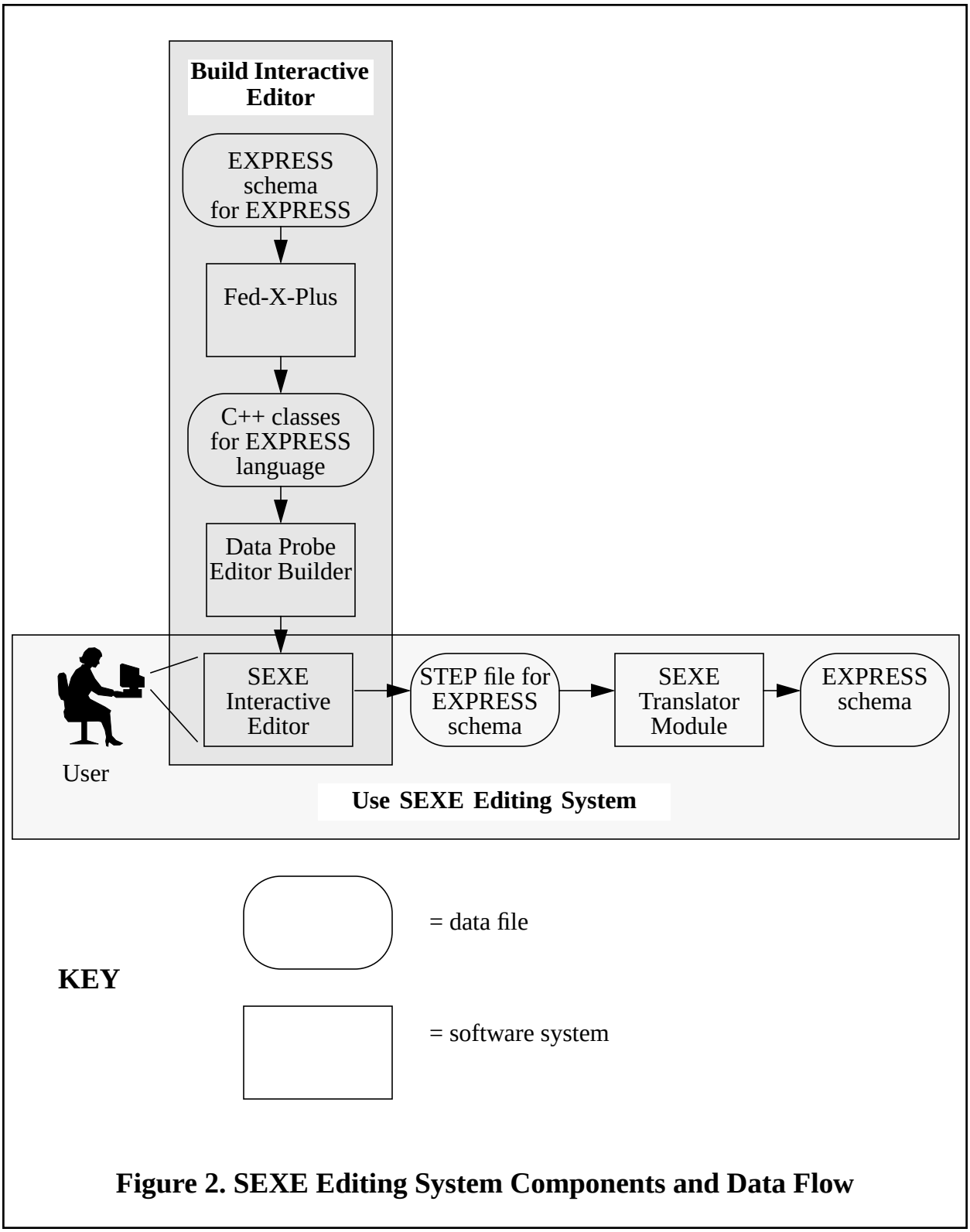

Figure 2. SEXE Editing System Components and Data Flow

The Entity Instance List window of the Interactive Editor, on the other hand, is not so congenial a part of an EXPRESS editor for two reasons. First, the name "Entity Instance List" tends to confuse the SEXE user. The user is engaged in writing entity definitions, type definitions, etc., not instances. This is a side effect of using a exchange file editor as an EXPRESS editor, of course, not a defect in the DPE Builder.

A second drawback of the Entity Instance List window is that it shows a print representation of the objects being edited in exchange file format, rather than in EXPRESS. For the example described in section 4.3, the appearance of the Entity Instance List window is almost identical to the data section of the file shown in Figure 3. This detracts significantly from the suitability of the Interactive Editor for writing EXPRESS. As mentioned earlier, however, the identifying number and type name of each object are displayed. In addition since the first attribute of most EXPRESS objects underlying the Interactive Editor (type definitions, entity definitions, etc.) is a string giving the name of the object, and strings get printed in human-readable form in exchange files, the name of each object is visible to the user. These three data items make it fairly easy to navigate around the Entity Instance List window, despite it being in an awkward format.

4.2 Translation Module

The SEXE Translation Module reads in an exchange file representing one or more EXPRESS schemas and writes out an EXPRESS file for the schemas. The Translation Module is written in Common LISP.

4.2.1 Using the Translation Module

To translate an exchange file for an EXPRESS schema into EXPRESS, a LISP environment containing the Translation Module software is started up. Then the LISP command **(print-express *"step-file-name" "express-file-name"*)** is executed. The file whose name is *step-file-name* is the file generated from the Interactive Editor and must exist beforehand. The file whose name is *express-file-name* gets created and contains the resulting EXPRESS. The file shown in Figure 4 was created this way.

Because of the way print-express works, the blocks (and other components) that make up one schema may be reused to define other schemas in the same file. If, for example, one schema has been built, and it is desired to define a second schema which contains a subset of the entities in the first schema, it suffices to define a single additional schema object in which the list of blocks contains only the desired subset. The Translation Module will print out both schemas in full in the same file.

4.2.2 LISP Functions in Translation Module

The Translation Module is built on top of a set of utilities for reading, processing, and writing exchange files developed a few years ago by one of the authors. An additional 25 functions which print EXPRESS were written specifically for the Translation Module. Nine of these print stubs (either EXPRESS definitions that have no details or comments in EXPRESS file format). In order to use the utilities in the Translation

Module, it was necessary also to write a table (in a specific LISP-readable format) that gives the names of all attributes for each entity in EXPRESS for SEXE which has attributes and may be instantiated.

4.2.3 Translation Module in C

It would be feasible to rewrite the Translation Module in C using the NIST PDES Toolkit [Clark90] (with minor modifications to allow forward references) in place of the LISP facilities mentioned above for handling exchange files. The original implementation was done in LISP only because the author who did it can write LISP about three times as fast as C. If the SEXE prototype is developed further, it would be a good idea to write a C version of the Translation Module. Most computer systems which might run the SEXE editing system have access to a C compiler, but Common LISP is not commonly available. Also, it might be feasible to have the Interactive Editor print EXPRESS by making a very simple change to the DPE Builder, if a C version of the Translation Module were available.

4.3 An Example

The Interactive Editor was used to build a small schema including seven geometry entities, which were printed to an exchange file by the Editor. The exchange file is shown in Figure 3. That file was put through the Translation Module which output the EXPRESS file which is shown in Figure 4.

```
STEP;
HEADER;
ENDSEC;
DATA;
@1=SCHEMA_BLOCK('geometry_schema',,,,(#2,#3,#4,#5,#6,#7,#8));
@2=ENTITY_BLOCK('geometry',,,,(#9),,,,);
@3=ENTITY_BLOCK('coordinate_system',,(#2),(#10,#11),,,,);
@4=ENTITY_BLOCK('point',#28,(#2),(),,,,);
@5=ENTITY_BLOCK('cartesian_point',,(#4),(#12,#13,#14),,,,);
@6=ENTITY_BLOCK('vector',#31,(#2),(),,,,);
@7=ENTITY_BLOCK('direction',,(#6),(#15,#16,#17),,,,);
@8=ENTITY_BLOCK('transformation',,(#2),(#18,#19,#20,#21,#22),,,,);
@9=EXPLICIT_ATTR(('local_coordinate_system'),.T.,#23);
@10=EXPLICIT_ATTR(('reference_coordinate_system'),.T.,#23);
@11=EXPLICIT_ATTR(('axis_set'),,#24);
@12=EXPLICIT_ATTR(('x_coordinate'),,#25);
@13=EXPLICIT_ATTR(('y_coordinate'),,#25);
@14=EXPLICIT_ATTR(('z_coordinate'),.T.,#25);
@15=EXPLICIT_ATTR(('x'),,#25);
@16=EXPLICIT_ATTR(('y'),,#25);
@17=EXPLICIT_ATTR(('z'),.T.,#25);
@18=EXPLICIT_ATTR(('axis1'),.T.,#26);
@19=EXPLICIT_ATTR(('axis2'),.T.,#26);
@20=EXPLICIT_ATTR(('axis3'),.T.,#26);
@21=EXPLICIT_ATTR(('local_origin'),.T.,#27);
@22=EXPLICIT_ATTR(('scale'),.T.,#25);
@23=ENTITY_BLOCK_WRAPPER(#3);
@24=ENTITY_BLOCK_WRAPPER(#8);
@25=SIMPLE_TYPES(.STRING_REAL.);
@26=ENTITY_BLOCK_WRAPPER(#7);
@27=ENTITY_BLOCK_WRAPPER(#5);
@28=SUPERTYPE_DECLARATION(,#29);
@29=SUPERTYPE_EXPRESSION(#30,());
@30=SUPERTYPE_ENTITY_REF(#5);
@31=SUPERTYPE_DECLARATION(,#32);
@32=SUPERTYPE_EXPRESSION(#33,());
@33=SUPERTYPE_ENTITY_REF(#7);
ENDSEC;
ENDSTEP;
```

Figure 3. STEP File EXPRESS Schema for Geometry

```
SCHEMA geometry_schema;

ENTITY geometry;
 local_coordinate_system: OPTIONAL coordinate_system;
END_ENTITY;

ENTITY coordinate_system
SUBTYPE OF (geometry);
 reference_coordinate_system: OPTIONAL coordinate_system;
 axis_set: transformation;
END_ENTITY;

ENTITY point
SUPERTYPE OF (cartesian_point )
SUBTYPE OF (geometry);
END_ENTITY;

ENTITY cartesian_point
SUBTYPE OF (point);
 x_coordinate: REAL;
 y_coordinate: REAL;
 z_coordinate: OPTIONAL REAL;
END_ENTITY;

ENTITY vector
SUPERTYPE OF (direction )
SUBTYPE OF (geometry);
END_ENTITY;

ENTITY direction
SUBTYPE OF (vector);
 x: REAL;
 y: REAL;
 z: OPTIONAL REAL;
END_ENTITY;

ENTITY transformation
SUBTYPE OF (geometry);
 axis1: OPTIONAL direction;
 axis2: OPTIONAL direction;
 axis3: OPTIONAL direction;
 local_origin: OPTIONAL cartesian_point;
 scale: OPTIONAL REAL;
END_ENTITY;

END_SCHEMA;
```

Figure 4. EXPRESS Schema for Geometry

5 Alternative Representations of EXPRESS

One of the most conceptually interesting parts of building the SEXE editing system was having the opportunity to construct alternative representations of EXPRESS, in which it is built in terms of itself. Four alternative representations were built. They are shown in appendices C (an EXPRESS schema we will call “EXPRESS for SEXE”), D (another EXPRESS schema, which we will call “Algorithmic EXPRESS”), E (an exchange file we will call “Exchange File EXPRESS”), and F (a LISP property list we will call “LISP EXPRESS”).

The formal definition of EXPRESS is given in the EXPRESS reference manual [Spiby], where a production language similar to BNF is used. This definition is shown in its entirety in Appendix B. The production rules in Appendix B have two different functions: to describe productions in terms of tokens, and to show how the tokens are represented with ASCII characters.

Because the work reported here was a prototyping effort, a subset of EXPRESS was selected so the work could be done quickly. All four of the representations in the appendices define the same subset of EXPRESS. The rules which define the subset are marked in Appendix B. This subset defines the following basic features of EXPRESS: schemas, entities, data types (including simple types, aggregation types, defined types, select types, and enumeration types), and subtyping. The subset does not define: interface specifications, rules, functions, other algorithms, or expressions. It appears to be feasible to extend these alternate representations to cover all of EXPRESS. No known impediment is created by the different formats.

Three of the alternative representations are automatically interconvertible, as follows: the SEXE Translation Module can read in the Exchange File EXPRESS file (actually building another LISP-readable file in the process, not shown in the appendices), create an internal representation whose print representation is the LISP EXPRESS file, resolve the references in the internal representation, and print out the EXPRESS for SEXE file (without the comments).

EXPRESS for SEXE and Algorithmic EXPRESS were created with different objectives. The objective in EXPRESS for SEXE was to create a schema which would result in a friendly and powerful Interactive Editor when the schema was fed into the DPE Builder. The objective in Algorithmic EXPRESS was to demonstrate that a version of EXPRESS in EXPRESS could be created by applying strict algorithmic conversion methods to the production rules which define EXPRESS. Details of the differences are given below.

Other researchers have been interested in writing EXPRESS models of EXPRESS which make clear the semantic content of EXPRESS, not its syntactic structure. None of the alternate representations given here was constructed with this motivation, and none is claimed to have semantic clarity.

EXPRESS for SEXE and Algorithmic EXPRESS pass through the NIST Fed-X EXPRESS parser without error or warning. Both have two parts: type definitions and entity definitions. Each part is arranged alphabetically.

Discussions of the four versions follow.

5.1 EXPRESS Schema for Building the Interactive Editor

EXPRESS for SEXE (Appendix C) was written using a text editor. It was constructed to be usable by the DPE Builder and to result in an editor that is easier to use than one built using a schema which copies the production rules of the EXPRESS manual literally (such as Algorithmic EXPRESS). The guidelines used in writing EXPRESS for SEXE are:

1. Do not define the following keywords that are not needed to distinguish structure: ANDOR, END_ENTITY, END_SCHEMA, END_TYPE, ENTITY, OF, ONEOF, SCHEMA, SUBTYPE, SUPERTYPE, TYPE, WHERE, and do not define the following keywords which can be replaced by boolean flags: ABSTRACT, OPTIONAL, UNIQUE.

2. Do not define punctuation. None of it is needed to distinguish structure.

3. Combine several rules into a single rule, if this will improve understandability. To define entity, for example, there are production rules in Appendix B for “entity_block”, “entity_head”, and “entity_body”. If a structure editor incorporated these rules separately, it would be necessary to define the components (head and body) as well as the whole each time an entity instance was built. In EXPRESS for SEXE the “entity_block” is defined to include all the components of an “entity_head” and an “entity_body”. “entity_head” and “entity_body” are not defined at all, and when an instance of an entity is to be built using the editor, those components are not built.

4. Do not distinguish between an object and a reference to an object. That is not necessary in EXPRESS. For example, where the production rules have both “entity_block” and “entity_ref”, only “entity_block” is needed in EXPRESS.

5. Use subtyping wherever possible, because the DPE Builder handles subtyping very nicely. Do not have any entity be the subtype of more than one supertype, however, because the DPE Builder cannot handle that situation.

6. Do not use “select” statements. An editor built by the DPE Builder does not check typing on select statements.

7. Represent EXPRESS keywords for simple data types and aggregation names by enumeration constants formed by adding the prefix “string_” to the keyword.

8. Represent any numbered language rule of the production rules shown in Appendix B from 140 through the end[1] that is to be included in EXPRESS for SEXE by an entity definition.

The prohibition of rule five against multiple supertypes makes an awkwardness in EXPRESS for SEXE. Entity_block and type_decl are both defined as subtypes of block. They should also be subtypes of named_types, but this would violate rule 5.

1. Rules 140 through the end contain (almost) nothing but productions consisting of tokens and punctuation characters.

Therefore, two otherwise unneeded subtypes of named_types have been defined: entity_block_wrapper and type_decl_wrapper. An entity_block_wrapper wraps an entity_block and a type_decl_wrapper wraps a type_decl.

Rules 6 and 7 lead to an interesting contrast between EXPRESS for SEXE and Algorithmic EXPRESS. EXPRESS for SEXE has lots of subtyping and no select statements, while Algorithmic EXPRESS has lots of select statements but no subtyping. The two constructions are alternatives, as long as supertypes do not have attributes, which is the case in EXPRESS for SEXE.

A “bootstrap kernel” may be defined for this schema. The kernel is a subset of the statements in the schema. For the subset to be called a bootstrap kernel:

- The statements in the subset must form a well-defined schema (i.e., the statements must form a sub-schema), and
- It must be possible to define the full schema in terms of the subset.

The statements needed for a bootstrap kernel are marked in Appendix B with a “K” in the comment given just before each statement. An attempt has been made to make the bootstrap kernel as small as possible. It would be feasible to show that the marked statements form a bootstrap kernel by deleting all the other statements, using the resulting schema as input to the DPE Builder to build an editor, and using the editor to define the full schema. This has not been done.

5.2 Algorithmically Derived EXPRESS

Algorithmic EXPRESS (Appendix D) defines EXPRESS in EXPRESS by sticking as closely as possible to the production rules for EXPRESS given in Appendix B. A set of algorithms for making the translation was defined and applied by the authors as though we were a computer, with the thought that a machine translator from production rules to EXPRESS might be feasible. The motivation for building Algorithmic EXPRESS was purely the intellectual challenge of determining whether such a thing was possible, by trying to do it. Changes from the EXPRESS manual are made to carve the subset out cleanly and to accommodate the differences between EXPRESS and the production language in which Appendix B is written. The algorithms are:

1. Represent EXPRESS keywords in terms of characters in EXPRESS by using TYPE statements. The name of the type is formed by adding the prefix “token_” to the keyword. The type itself is an enumeration of one item which is formed by adding the prefix “string_” to the keyword. It would be nice to do without the prefix, but since these are keywords, they cannot legally be used verbatim.

2. Tokens in the production rules of Appendix B which are not EXPRESS keywords will be used verbatim.

3. Punctuation marks in the production rules cannot legally be used in EXPRESS, so we use their English names, preceded by the prefix “token_”, and define them as TYPES.

4. A production rule of the form aa = bb, where bb is a TYPE, becomes:

```
TYPE aa = bb;
END_TYPE;
```

5. A production rule of the form aa = bb | cc | dd ... becomes

```
TYPE aa = SELECT (bb, cc, dd, ...);
END_TYPE;
```

6. A production rule of the form aa = bb {cc} [dd] becomes:

```
ENTITY aa;
   bb_at : bb;
   ccs : LIST [0:?] OF cc;
   dd_at: OPTIONAL dd;
END_ENTITY;
```

Several sub-algorithms apply here:

A. For a production rule entry of the form bb, the name of the attribute representing the entry in the definition of the entity representing the rule is formed by adding the suffix "_at" and the type of the attribute is bb. If this would make two attributes of the same entity have the same name, the second one will have the suffix "_at2".

B. For a production rule entry of the form {cc}, the name of the attribute is formed by adding the suffix "s", and the type of the attribute is LIST [0:?] OF cc.

C. For a production rule entry of the form [dd], the name and type of the attribute are as in A, but the attribute is optional.

7. A construct of the form { aa | bb } appearing in a production rule will be represented by defining a select type. The name of the type will be select_N, where N is an integer whose initial value will be 1 that will be incremented by 1 each time it is necessary to make another such "select_N" definition. Two such definitions are in Algorithmic EXPRESS.

8. A construct of the form {aa bb cc} or [aa bb cc] appearing in a production rule will be represented by defining an entity named construct_M as though there were a rule: construct_M = aa bb cc and the form {construct_M} or [construct_M] appeared in the production. M is an integer whose initial value will be 1 that will be incremented by 1 each time it is necessary to make another such "construct_M" definition. Eleven such definitions are in Algorithmic EXPRESS.

An editor using this schema was built with the DPE Builder. The editor that resulted is extremely cumbersome, because to build an entity or type, a complex hierarchy of tedious individual components must be defined. The complexity is a result of the production rules being individually small but forming a complex hierarchy. The

tediousness is a result of explicitly defining all the keywords and punctuation. Automatic translation into EXPRESS from the exchange file resulting from using the editor has not been done, but should be quite easy to implement because all elements of EXPRESS are explicitly modelled in this schema.

5.3 STEP Exchange File

Every STEP file has an underlying EXPRESS schema which tells how to interpret the file.

Exchange File EXPRESS (Appendix E) is an exchange file which defines EXPRESS. The EXPRESS schema underlying this file is EXPRESS for SEXE, which also defines EXPRESS. The Exchange File EXPRESS file was built by using the SEXE Interactive Editor to define a structure conceptually identical to EXPRESS for SEXE. Since the Interactive Editor prints out exchange files, this conceptual structure is in an exchange file. The SEXE Translation Module, as noted earlier, will translate the Exchange File EXPRESS file into the EXPRESS for SEXE file.

For the authors, the Exchange File EXPRESS file has a subjectively strange, warped feel to it. Here it seems that EXPRESS has been convoluted with both itself and the rules for exchange files.

There is no need for this file in the SEXE editing system. It was constructed as a curiosity.

5.4 LISP-Readable Property List File

LISP EXPRESS (Appendix F) is a LISP-readable copy of Exchange File EXPRESS. Its basic nature is identical to that of Exchange File EXPRESS. There are minor differences in punctuation and form. The closest thing to a major difference is that LISP EXPRESS includes the names of the attributes while Exchange File EXPRESS does not. Our intent in including LISP EXPRESS is simply to show yet another representation of EXPRESS.

LISP EXPRESS is a file that may be read into Common LISP by simply LISP “load”ing the file. When the file is loaded, an internal data structure is automatically built. The file was built automatically from Exchange File EXPRESS, using LISP utilities mentioned earlier.

6 Duality of Entities and Instances

Having schemas in exchange files as exemplified in the SEXE editing system demonstrates the duality of layers in information modeling. Instances of data description objects may be created in one layer and then be used as terms for defining instances in the next lower layer.

In the STEP standards arena, there are only two layers of information representation, the EXPRESS layer and the exchange file layer. These two layers have different formats. It would be interesting to develop a modeling language that could be used for both layers. Preferably, such a language would support an arbitrary number of layers, not just two.

Even the division into layers is arbitrary. The EXPRESS and exchange file layers could be compacted into a single layer, for example, by representing the information of an exchange file in an EXPRESS schema by subtyping. For example, using the schema for geometry of Figure 4, we might have the following data section in an exchange file that defines but one instance:

```
DATA
@1 = cartesian_point ( ,1,2,3);
ENDSEC
```

The same information could be represented by adding one entity to the schema:

```
ENTITY instance1
SUBTYPE OF cartesian_point;
WHERE
    NOT (EXISTS (local_coordinate_system));
    x_coordinate = 1;
    y_coordinate = 2;
    z_coordinate = 3;
END_ENTITY;
```

An information modeling language which could handle these issues smoothly would be an impressive development.

7 Conclusions

This paper has presented SEXE, a structural editor for EXPRESS which consists of a structure editor (built using an exchange file editor builder) plus a translation module. The input to the editor builder is an EXPRESS schema, so we have discussed alternate EXPRESS schemas for EXPRESS (as well as other representations of EXPRESS) and assessed the characteristics of the alternates which make them suitable or unsuitable as schemas for building an editor. Finally, we have discussed some issues in information modeling highlighted by building SEXE.

Although we have not defined the entire EXPRESS language in EXPRESS, we have learned that it is feasible to write widely varied EXPRESS schemas for EXPRESS. The form of a given EXPRESS schema for EXPRESS is driven by the intended uses of the schema. The schema for Algorithmic EXPRESS (defined to see if algorithmic derivation is feasible) is quite different from the EXPRESS for SEXE schema (defined to produce a friendly editor when fed to the DPE Builder).

7.1 Assessment of the SEXE Editing System

What is easy or hard, friendly or unfriendly, is very subjective, so this section is open to disagreement.

7.1.1 Good Things

Usability - The SEXE editing system is clearly usable for writing EXPRESS (see details below). It is moderately friendly.

Coverage of EXPRESS - Although it can currently be used for creating only a subset of EXPRESS, it would be feasible to extend coverage to all of EXPRESS.

Correctness - The SEXE editing system writes faultless EXPRESS.

Labor Saving Device - The SEXE editing system saves the user from having to learn the details of the grammar and syntax of EXPRESS.

7.1.2 Bad Things

Usability - The SEXE editing system is cumbersome. Even though the schema underlying the SEXE editing system was written with conscious effort to combine related language elements, the SEXE editing system still has too many small components of structure and not enough large ones (see details below).

Coverage of EXPRESS - The SEXE editing system can currently be used for creating only a subset of EXPRESS.

Viewing EXPRESS - As mentioned earlier, the Interactive Editor does not print actual EXPRESS in the Entity Instance List window (the logical place for it). No EXPRESS is visible to the user until the Translation Module is run.

Information Linkage - The association between an entity instance and any attribute values which are also entity instances is not clear enough, particularly if the type of the attribute value shown in a SEE window is a supertype of the value actually entered (or desired to be entered).

Missing Functionality - The SEXE editing system cannot read in an existing EXPRESS file for editing unless the file was built on the SEXE editing system and the exchange file version is still available.

7.1.3 Details

The SEXE editing system might be useful as a teaching tool for people learning EXPRESS. It is easy to start writing EXPRESS with the SEXE editing system. Even if you know nothing about EXPRESS, you can get an idea of what it is by playing with the editor, and you can write a schema after playing a while. With a text editor, if you know nothing about EXPRESS, you cannot write anything meaningful.

The Interactive Editor is at its most cumbersome when it is used for structures which are deep hierarchies of small pieces. The only current case of this is in supertype statements. If the schema for EXPRESS underlying the SEXE editing system included the definitions necessary to describe EXPRESS functions, using the Interactive Editor to define functions would be cumbersome for the same reason. We suspect most users

would not tolerate writing functions in the SEXE editing system (or any structure editor) for very long. An example of a simple expression in structural form is given in [Rösch] which makes this point. The problem with supertype statements could be largely solved by defining the most common types of supertype statements as separate subtypes of supertype statement in EXPRESS for SEXE.

7.2 Recommendations for Building EXPRESS Editors

A brief presentation of issues regarding EXPRESS editors is given in [Spiby91b].

The current method used commonly to write EXPRESS - generate it on a text editor, then run through a parser - is fairly efficient for those who are familiar with EXPRESS.

The EMACS text editor, modified by adding an EXPRESS mode, might be adequate. As noted earlier, such a thing has been developed by researchers outside NIST, but we have not used it.

It seems unlikely that a pure structure editor could do better than EMACS with an EXPRESS mode. Doing better would require a sophisticated, hand-crafted editor combining features of structure editing with features of text editing. For example, entities might be created in structure mode, while functions are written in text mode. It would be very useful to have both continuous and intermittent syntax checking while working in text mode. In the structure mode, the connections between entities should be made more visible than in the SEXE editing system. For example, SEE windows might be connected by lines to form networks. It would be useful to be able to zoom in (for editing) and out (for viewing) on the pictures of networks. It would be desirable to give the user a choice between structure editing and text editing, just as good interactive systems give the user a choice of mouse or keyboard. Knowledgeable users could then work primarily in text mode, while neophytes would work primarily in structure mode. As noted earlier, the EXPRESS editor built by Mike Yinger has most of these characteristics, but we have not used it.

The SEXE editing system could be upgraded substantially so that it is a practical working tool rather than a prototype. This would require that the following be done:

1. Complete the DPE Builder software.

2. Write an improved version of EXPRESS for SEXE to be used as input to the DPE Builder.

3. Rewrite the Translation Module software. It is tailored to EXPRESS for SEXE, so changes to the schema force changes to the Translation Module.

4. Write a users manual.

5. Write an EXPRESS to exchange file translator so EXPRESS files may be read in.

7.3 Ideas for Future Related Work

Some opportunities for future work are:

- Refine the EXPRESS for SEXE schema further (even without extending it) so that the Interactive Editor would be less cumbersome. As mentioned earlier, improving the handling of supertype statements in this manner is feasible.
- Extend the coverage of the SEXE editing system to the entire EXPRESS language. This is clearly feasible. Doing it in a way that would result in a non-cumbersome Interactive Editor being built by the DPE Builder would be a challenge.
- Implement in software the algorithms defined in this paper for deriving Algorithmic EXPRESS from production rules. The software would read production rules in the language used for the EXPRESS reference manual and write EXPRESS. This appears to be straightforward and should not take long, if done in a high-level language such as LISP.
- Improve the user interface. As noted earlier, the SEXE Interactive Editor would be much improved if it wrote EXPRESS in what is currently called the Entity Instance List Window (a name that would be inappropriate if EXPRESS were displayed in the window). This could be done by developing a new component for the DPE Builder software, rewriting the SEXE Translator Module in C++, and making the Translator much more sophisticated (since it would have to be able to deal with incomplete definitions). This is quite difficult.
- Modify the DPE Builder more generally, so that C++ code segments (written to implement domain-specific features in the editor ultimately produced by the Builder) could be written by a systems developer and automatically incorporated by the Builder in the code for the editor. This modification is not specific to the job of writing EXPRESS and is also very difficult.
- Build a mixed-mode structure-or-text editor for EXPRESS (or other languages). Developing sound principles that relate the behavior of the editor to the characteristics of the language being edited is a challenge.

A References

[Altemueller88a] Altemueller, J.; *The STEP File Structure*; ISO TC184/SC4/WG1 Document N279; September, 1988

[Altemueller88b] Altemueller, J.; *Mapping from Express to Physical File Structure*; ISO TC184/SC4/WG1 Document N280; September, 1988

[Clark90] Clark, Stephen N.; *An Introduction to the NIST PDES Toolkit;* NISTIR 4336; National Institute of Standards and Technology; Gaithersburg, MD; May, 1990

[Clark92] Clark, Stephen N.; and Libes, Don; *Fed-X: The NIST Express Translator*; NISTIR 4371; National Institute of Standards and Technology; Gaithersburg, MD; Revised January, 1992

[ISO] International Organization for Standardization; committee draft STEP Part 1 *Overview and Fundamental Principles*; ISO TC184/SC4 Document N134; May 15, 1992

[Jasnoch] Jasnoch, Uwe; et al; "Xpresso - An Object-Oriented Graphic-Interactive Design Environment for EXPRESS"; in conference notes from *EXPRESS User's Group - EUG '9*1; Houston, Texas; October 1991

[Kramer] Kramer, Thomas R., *A Library of Material Removal Shape Element Volumes (MRSEVs)*; NISTIR 4809; National Institute of Standards and Technology; Gaithersburg, MD; March, 1992

[Linton] Linton, Mark A.; Calder, Paul R.; and Vlissides, John M.; *InterViews: A C++ Graphical Interface Toolkit*; Technical Report CSL-TR-88-358; Stanford University; July, 1988

[Morris90] Morris, Katherine C.; and McLay, Michael J; *The NIST STEP Class Library (STEP into the Future)*; NISTIR 4411; National Institute of Standards and Technology; Gaithersburg, MD; August, 1990

[Morris91] Morris, Katherine C.; *Architecture for the Validation Testing System Software*; NISTIR 4742, National Institute of Standards and Technology; Gaithersburg, MD; December 1991

[Rösch] Rösch, P.; "Exploit - a Graphical Editor for EXPRESS-G"; in conference notes from *EXPRESS User's Group - EUG '9*1; Houston, Texas; October 1991

[Spiby91a] Spiby, Philip; draft STEP Part 11 *EXPRESS Language Reference Manual*; ISO TC184/SC4/WG5 Document N14; April 29, 1991

[Spiby91b] Spiby, Philip; "An EXPRESS Development Environment"; in conference notes from *EXPRESS User's Group - EUG '9*1; Houston, Texas; October 1991

[Van Maanen] Van Maanen, Jan; draft STEP Part 21 *Clear Text Encoding of the Exchange Structure*; ISO TC184/SC4; April 1992

[Wilson] Wilson, Peter R.; *Processing Tools for EXPRESS;* Rensselaer Design Research Center Technical Report No: 92003; February, 1992

[Yinger] Yinger, Michael A.; “Implementing an EXPRESS Modeling Environment: Smalltalk Meets EXPRESS”; in conference notes from *EXPRESS User’s Group - EUG ‘9*1; Houston, Texas; October 1991

B Production Rules for EXPRESS

This is a copy of Annex A1 of the EXPRESS Reference Manual. It contains a production rule definition of the entire EXPRESS language (not just a subset). The rule numbers for the subset covered explicitly by Algorithmic EXPRESS are in **boldface and underlined**. Rule numbers for stub definitions in Algorithmic EXPRESS are in ***italic boldface and underlined***.

0 | ABS = ‘abs’ .
1 | ABSTRACT = ‘abstract’ .
2 | ACOS = ‘acos’ .
3 | AGGREGATE = ‘aggregate’ .
4 | ALIAS = ‘alias’ .
5 | AND = ‘and’ .
6 | ANDOR = ‘andor’ .
7 | ARRAY = ‘array’ .
8 | AS = ‘as’ .
9 | ASIN = ‘asin’ .

10 | ATAN = ‘atan’ .
11 | BAG = ‘bag’ .
12 | BEGIN = ‘begin’ .
13 | BINARY = ‘binary’ .
14 | BLENGTH = ‘blength’ .
15 | BOOLEAN = ‘boolean’ .
16 | BY = ‘by’ .
17 | CASE = ‘case’ .
18 | CONST_E = ‘const_e’ .
19 | CONSTANT = ‘constant’ .

20 | CONTEXT = ‘context’ .
21 | COS = ‘cos’ .
22 | DERIVE = ‘derive’ .
23 | DIV = ‘div’ .
24 | ELSE = ‘else’ .
25 | END = ‘end’ .
26 | END_ALIAS = ‘end_alias’ .
27 | END_CASE = ‘end_case’ .
28 | END_CONSTANT = ‘end_constant’ .
29 | END_CONTEXT = ‘end_context’ .

30 | END_ENTITY = ‘end_entity’ .
31 | END_FUNCTION = ‘end_function’ .
32 | END_IF = ‘end_if’ .
33 | END_LOCAL = ‘end_local’ .
34 | END_MODEL = ‘end_model’ .
35 | END_PROCEDURE = ‘end_procedure’ .

36 | END_REPEAT = ‘end_repeat’ .
37 | END_RULE = ‘end_rule’ .
38 | END_SCHEMA = ‘end_schema’ .
39 | END_TYPE = ‘end_type’ .

40 | ENTITY = ‘entity’ .
41 | ENUMERATION = ‘enumeration’ .
42 | ESCAPE = ‘escape’ .
43 | EXISTS = ‘exists’ .
44 | EXP = ‘exp’ .
45 | FALSE = ‘false’ .
46 | FIXED = ‘fixed’ .
47 | FOR = ‘for’ .
48 | FORMAT = ‘format’ .
49 | FROM = ‘from’ .

50 | FUNCTION = ‘function’ .
51 | GENERIC = ‘generic’ .
52 | HIBOUND = ‘hibound’ .
53 | HIINDEX = ‘hiindex’ .
54 | IF = ‘if’ .
55 | IN = ‘in’ .
56 | INSERT = ‘insert’ .
57 | INTEGER = ‘integer’ .
58 | INVERSE = ‘inverse’ .
59 | LENGTH = ‘length’ .

60 | LIKE = ‘like’ .
61 | LIST = ‘list’ .
62 | LOBOUND = ‘lobound’ .
63 | LOINDEX = ‘loindex’ .
64 | LOCAL = ‘local’ .
65 | LOG = ‘log’ .
66 | LOG10 = ‘log10’ .
67 | LOG2 = ‘log2’ .
68 | LOGICAL = ‘logical’ .
69 | MOD = ‘mod’ .

70 | MODEL = ‘model’ .
71 | NOT = ‘not’ .
72 | NUMBER = ‘number’ .
73 | NVL = ‘nvl’ .
74 | ODD = ‘odd’ .
75 | OF = ‘of’ .
76 | ONEOF = ‘oneof’ .
77 | OPTIONAL = ‘optional’ .

78 | OR = ‘or’ .
79 | OTHERWISE = ‘otherwise’ .

80 | PI = ‘pi’ .
81 | PROCEDURE = ‘procedure’ .
82 | QUERY = ‘query’ .
83 | REAL = ‘real’ .
84 | REFERENCE = ‘reference’ .
85 | REMOVE = ‘remove’ .
86 | REPEAT = ‘repeat’ .
87 | RETURN = ‘return’ .
88 | ROLESOF = ‘rolesof’ .
89 | RULE = ‘rule’ .

90 | SCHEMA = ‘schema’ .
91 | SELECT = ‘select’ .
92 | SELF = ‘self’ .
93 | SET = ‘set’ .
94 | SIN = ‘sin’ .
95 | SIZEOF = ‘sizeof’ .
96 | SKIP = ‘skip’ .
97 | SQRT = ‘sqrt’ .
98 | STRING = ‘string’ .
99 | SUBTYPE = ‘subtype’ .

100 | SUPERTYPE = ‘supertype’ .
101 | TAN = ‘tan’ .
102 | THEN = ‘then’ .
103 | TO = ‘to’ .
104 | TRUE = ‘true’ .
105 | TYPE = ‘type’ .
106 | TYPEOF = ‘typeof’ .
107 | UNIQUE = ‘unique’ .
108 | UNKNOWN = ‘unknown’ .
109 | UNTIL = ‘until’ .

110 | USEDIN = ‘usedin’ .
111 | USE = ‘use’ .
112 | VALUE = ‘value’ .
113 | VAR = ‘var’ .
114 | WHERE = ‘where’ .
115 | WHILE = ‘while’ .
116 | XOR = ‘xor’ .
117 | add_like_op = ‘+’ | ‘-’ | OR | XOR .
118 | binary_literal = ‘%’ bit { bit } .
119 | bit = ‘0’ | ‘1’ .

120 | character = digit | letter | special .
121 | digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ .
122 | digits = digit { digit } .
123 | integer_literal = digits .
124 | letter = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’ .
125 | logical_literal = FALSE | TRUE | UNKNOWN .
126 | lparen_not_star = ‘(‘ not_star .
127 | multiplication_like_op = ‘*’ | ‘/’ | DIV | MOD | AND .
128 | not_lparen_star = not_paren_star | ‘)’ .
129 | not_paren_star = letter | digit | not_paren_star_special .

130 | not_paren_star_special = ‘!’ | ‘@’ | ‘#’ | ‘$’ | ‘%’ | ‘^’ | ‘&’ | ‘_’ | ‘-’ | ‘+’ | ‘=’ | ‘{‘ | ‘}’ | ‘[‘ | ‘]’ | ‘~’ | ‘:’ | ‘;’ | ‘’’ | ‘””’ | ‘<‘ | ‘>’ | ‘,’ | ‘.’ | ‘?’ | ‘/’ | ‘|’ | ‘\’ .
131 | not_rparen = not_paren_star | ‘*’ | ‘(‘ .
132 | not_star = not_paren_star | ‘(‘ | ‘)’ .
133 | real_literal = digits ‘.’ [digits] [‘e’ [sign] digits] .
134 | rel_op = ‘<‘ | ‘>’ | ‘<=’ | ‘>=’ | ‘<>’ | ‘=’ | ‘:<>:’ | ‘:=:’ .
135 | sign = ‘+’ | ‘-’ .
136 | simple_id = letter { letter | digit | ‘_’ } .
137 | special = not_paren_star_special | ‘*’ | ‘(‘ | ‘)’ .
138 | star_not_rparen = ‘*’ not_rparen .
139 | string_literal = \q { character | \s | \o } \q .

140 | actual_parameter_list = ‘(‘ parameter { ‘,’ parameter } ‘)’ .
141 | aggregate_initializer = ‘[‘ [element { ‘,’ element }] ‘]’ .
142 | aggregate_source = expression .
143 | aggregate_type = AGGREGATE [‘:’ type_label] OF parameter_type .
144 | aggregation_types = array_type | bag_type | list_type | set_type .
145 | algorithm_head = { declaration } [constant_decl] [local_decl] .
146 | alias_id = simple_id .
147 | alias_ref = alias_id .
148 | alias_stmt = ALIAS alias_id FOR general_ref { qualifier } ‘;’ { stmt } END_ALIAS ‘;’ .
149 | array_type = ARRAY ‘[‘ bound_spec ‘]’ OF [OPTIONAL] [UNIQUE] base_type .

150 | assignment_stmt = general_ref { qualifier } ‘:=’ expression ‘;’ .
151 | attribute_decl = attribute_id | referenced_attribute .
152 | attribute_id = simple_id .
153 | attribute_qualifier = ‘.’ attribute_ref .
154 | attribute_ref = attribute_id .
155 | bag_type = BAG [bound_spec] OF base_type .
156 | base_type = aggregation_types | simple_types | named_types .
157 | binary_type = BINARY [‘(‘ width ‘)’ [FIXED]] .
158 | boolean_type = BOOLEAN .
159 | bound_1 = simple_expression .

160 | bound_2 = simple_expression .
161 | bound_spec = '[' bound_1 ':' bound_2 ']' .
162 | built_in_constant = CONST_E | PI | SELF | '?' .
163 | built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS | EXP |
FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND | LOINDEX | LOG |
LOG2 | LOG10 | NVL | ODD | ROLESOF | SIN | SIZEOF | SQRT | TAN | TYPEOF |
USEDIN | VALUE .
164 | built_in_procedure = INSERT | REMOVE .
165 | case_action = case_label { ',' case_label } ':' stmt .
166 | case_label = expression .
167 | case_stmt = CASE selector OF { case_action } [OTHERWISE ':' stmt] END_CASE ';' .
168 | compound_stmt = BEGIN { stmt } END ';' .
169 | conformant_aggregate = conformant_type [bound_spec] OF parameter_type .

170 | conformant_type = ARRAY | BAG | LIST | SET .
171 | constant_decl = CONSTANT { constant_body } END_CONSTANT ';' .
172 | constant_body = constant_id ':' base_type ':=' expression ';' .
173 | constant_factor = built_in_constant | constant_ref .
174 | constant_id = simple_id .
175 | constant_ref = constant_id .
176 | declaration = entity_block | function_block | procedure_block | type_decl .
177 | derived_attr = attribute_decl ':' base_type ':=' expression ';' .
178 | derive_clause = DERIVE { derived_attr } .
179 | element = expression [':' repetition] .

180 | embedded_remark = '(*' { not_lparen_star | lparen_not_star |
star_not_rparen | embedded_remark } '*)' .
181 | entity_block = entity_head entity_body END_ENTITY ';' .
182 | entity_body = { explicit_attr } [derive_clause] [inverse_clause]
[unique_clause] [where_clause] .
183 | entity_head = ENTITY entity_id [subsuper] ';' .
184 | entity_id = simple_id .
185 | entity_init = entity_ref '(' [expression { ',' expression }] ')' .
186 | entity_or_rename = entity_ref [AS entity_id] .
187 | entity_ref = entity_id .
188 | enumeration_id = simple_id .
189 | enumeration_ref = [type_ref '.'] enumeration_id .

190 | enumeration_type = ENUMERATION OF '(' enumeration_id { ',' enumeration_id } ')' .
191 | escape_stmt = ESCAPE ';' .
192 | explicit_attr = attribute_decl { ',' attribute_decl } ':' [OPTIONAL] base_type ';' .
193 | expression = simple_expression [rel_op_extended simple_expression] .
194 | factor = simple_factor ['**' simple_factor] .
195 | formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
196 | function_block = function_head algorithm_head { stmt } END_FUNCTION ';' .

197 | function_call = (built_in_function | function_ref) [actual_parameter_list] .
198 | function_head = FUNCTION function_id ['(' formal_parameter
{ ';' formal_parameter } ')'] ':' parameter_type ';' .
199 | function_id = simple_id .

200 | function_ref = function_id .
201 | general_ref = alias_ref | parameter_ref | variable_ref .
202 | generic_type = GENERIC [':' type_label] .
203 | group_qualifier = '\' entity_ref .
204 | if_stmt = IF expression THEN stmt { stmt } [ELSE { stmt }] END_IF ';' .
205 | increment = simple_expression .
206 | increment_control = variable_id ':=' bound_1 TO bound_2 [BY increment] .
207 | index = simple_expression .
208 | index_qualifier = '[' index ']' .
209 | initializer = aggregate_initializer | subsuper_init .

210 | integer_type = INTEGER .
211 | interface_specification = reference_clause | use_clause .
212 | interval = '{' bound_1 interval_op interval_item interval_op bound_2 '}' .
213 | interval_item = simple_expression .
214 | interval_op = '<' | '<=' .
215 | inverse_attr = attribute_id ':' [(SET | BAG) [bound_spec] OF]
entity_ref FOR attribute_ref ';' .
216 | inverse_clause = INVERSE { inverse_attr } .
217 | label = simple_id .
218 | labelled_attrib_list = [label ':'] referenced_attribute { ',' referenced_attribute } .
219 | labelled_expression = [label ':'] expression .

220 | list_type = LIST [bound_spec] OF [UNIQUE] base_type .
221 | literal = binary_literal | integer_literal | logical_literal | real_literal | string_literal .
222 | local_decl = LOCAL { local_variable } END_LOCAL ';' .
223 | local_variable = variable_id { ',' variable_id } ':' parameter_type [':=' expression] ';' .
224 | logical_expression = expression .
225 | logical_type = LOGICAL .
226 | named_types = entity_ref | type_ref .
227 | null_stmt = ';' .
228 | number_type = NUMBER .
229 | one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .

230 | parameter = expression .
231 | parameter_id = simple_id .
232 | parameter_ref = parameter_id .
233 | parameter_type = aggregate_type | conformant_aggregate | generic_type | base_type .
234 | precision_spec = simple_expression .
235 | procedure_block = procedure_head algorithm_head { stmt } END_PROCEDURE ';' .
236 | procedure_call_stmt = (built_in_procedure | procedure_ref) [actual_parameter_list] ';' .

```
237 | procedure_head = PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
        { ';' [ VAR ] formal_parameter } ')' ] ';' .
238 | procedure_id = simple_id .
239 | procedure_ref = procedure_id .

240 | qualifier = attribute_qualifier | group_qualifier | index_qualifier | subcomponent_qualifier .
241 | qualifiable_factor = attribute_ref | constant_factor | function_call | general_ref .
242 | qualified_attribute = SELF group_qualifier attribute_qualifier .
243 | query_expression = QUERY '(' variable_id '<*' aggregate_source '|'
        logical_expression ')' .
244 | real_type = REAL [ '(' precision_spec ')' ] .
245 | reference_clause = REFERENCE FROM schema_ref [ '(' resource_or_rename
        { ',' resource_or_rename } ')' ] ';' .
246 | referenced_attribute = attribute_ref | qualified_attribute .
247 | rel_op_extended = rel_op | IN | LIKE .
248 | remark = embedded_remark | tail_remark .
249 | rename_id = entity_id | function_id | procedure_id | type_id .

250 | repeat_control = [ increment_control ] [ while_control ] [ until_control ] .
251 | repeat_stmt = REPEAT repeat_control ';' { stmt } END_REPEAT ';' .
252 | repetition = simple_expression .
253 | resource_or_rename = resource_ref [ AS rename_id ] .
254 | resource_ref = constant_ref | entity_ref | function_ref | procedure_ref | type_ref .
255 | return_stmt = RETURN [ '(' expression ')' ] ';' .
256 | rule_block = rule_head algorithm_head { stmt } where_clause END_RULE ';' .
257 | rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')' ';' .
258 | rule_id = simple_id .
259 | schema_block = SCHEMA schema_id ';' schema_body END_SCHEMA ';' .

260 | schema_body = { interface_specification } [ constant_decl ] { declaration | rule_block } .
261 | schema_id = simple_id .
262 | schema_ref = schema_id .
263 | select_type = SELECT '(' named_types { ',' named_types } ')' .
264 | selector = expression .
265 | set_type = SET [ bound_spec ] OF base_type .
266 | simple_expression = term { add_like_op term } .
267 | simple_factor = enumeration_ref | initializer | interval | literal | qualifiable_factor
        { qualifier } | query_expression | '(' expression ')' | unary_op simple_factor .
268 | simple_types = binary_type | boolean_type | integer_type |
        logical_type | number_type | real_type | string_type .
269 | skip_stmt = SKIP ';' .

270 | stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt | escape_stmt |
        if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt | skip_stmt .
271 | string_type = STRING [ '(' width ')' [ FIXED ] ] .
272 | subcomponent_qualifier = '[' index ':' index ']' .
```

273 | subsuper = [supertype_declaration] [subtype_declaration] .
274 | subsuper_init = entity_init { '||' entity_init } .
275 | subtype_declaration = SUBTYPE OF '(' entity_ref { ',' entity_ref } ')' .
276 | supertype_declaration = (ABSTRACT SUPERTYPE) | ([ABSTRACT]
SUPERTYPE OF '(' supertype_expression ')') .
277 | supertype_expression = supertype_factor { (AND | ANDOR) supertype_factor } .
278 | supertype_factor = entity_ref | one_of | '(' supertype_expression ')' .
279 | tail_remark = '--' { \a | \s } \n .

280 | term = factor { multiplication_like_op factor } .
281 | type_decl = TYPE type_id '=' underlying_type ';' [where_clause] END_TYPE ';' .
282 | type_id = simple_id .
283 | type_label = simple_id .
284 | type_ref = type_id .
285 | unary_op = '+' | '-' | NOT .
286 | underlying_type = enumeration_type | select_type | aggregation_types |
simple_types | type_ref .
287 | unique_clause = UNIQUE labelled_attrib_list ';' { labelled_attrib_list ';' } .
288 | until_control = UNTIL expression .
289 | use_clause = USE FROM schema_ref ['(' entity_or_rename

290 | variable_id = simple_id .
291 | variable_ref = variable_id .
292 | where_clause = WHERE labelled_expression ';' { labelled_expression ';' } .
293 | while_control = WHILE expression .
294 | width = simple_expression .

C EXPRESS Schema Used to Build Editor

This is an EXPRESS file called “EXPRESS for SEXE” for a subset of the EXPRESS language. The schema defined here was used as input to the DPE Builder to build the SEXE Interactive Editor. Appendices D, E, and F define the same subset of EXPRESS as does this appendix. This schema is discussed in section 5.1 The comment above each EXPRESS statement contains the number of the production rule from which the statement was derived. The comment contains a “K” if the statement is part of the bootstrap kernel.

```
SCHEMA sexe_express;

(* K 7, 11, 61, 93 *)
TYPE aggregation_name =
 ENUMERATION OF (string_array, string_bag, string_list, string_set);
END_TYPE;

(* K 151 simplified *)
TYPE attribute_decl = STRING;
END_TYPE;

(* 193 simplified *)
TYPE expression = STRING;
END_TYPE;

(* 217 simplified *)
TYPE label = STRING;
END_TYPE;

(* K 266 simplified *)
TYPE simple_expression = STRING;
END_TYPE;

(* K 136 simplified *)
TYPE simple_id = STRING;
END_TYPE;

(* K 13, 15, 57, 68, 72, 83, 98, 157 simplified, 158, 210, 225, 228, 244 simplified, 271 simplified
*)
TYPE simple_type_name = ENUMERATION OF (string_binary, string_boolean, string_integer,
                                        string_logical, string_number, string_real, string_string);
END_TYPE;
```

```
(* K 143, 149, 155, 220, 265 *)
ENTITY aggregation_types
SUBTYPE OF (base_type);
 aggregation_type_name:     aggregation_name;
 bound_spec_entity:         OPTIONAL bound_spec;
 is_optional:               OPTIONAL BOOLEAN;
 is_unique:                 OPTIONAL BOOLEAN;
 base_type_entity:          base_type;
END_ENTITY;

(* K 156 *)
ENTITY base_type
ABSTRACT SUPERTYPE OF (ONEOF (aggregation_types, simple_types, named_types))
SUBTYPE OF (underlying_type);
END_ENTITY;

(* K Not in rules *)
ENTITY block
ABSTRACT SUPERTYPE OF (ONEOF (entity_block, rule_block, type_decl,
 procedure_block, function_block));
END_ENTITY;

(* K 159, 160, 161 *)
ENTITY bound_spec;
 bound_1:                   simple_expression;
 bound_2:                   simple_expression;
END_ENTITY;

(* 171 stub *)
ENTITY constant_decl;
END_ENTITY;

(* 178 stub *)
ENTITY derive_clause;
END_ENTITY;

(* K 181, 182, 183, 273, 275 *)
ENTITY entity_block
SUBTYPE OF (block);
 entity_id:                 simple_id;
 supertype_declaration_entity:OPTIONAL supertype_declaration;
 subtype_declaration:       OPTIONAL LIST [1:?] OF entity_block;
 explicit_attrs:            LIST [0:?] OF explicit_attr;
 derive_clause_entity:      OPTIONAL derive_clause;
 inverse_clause_entity:     OPTIONAL inverse_clause;
 unique_clause_entity:      OPTIONAL unique_clause;
 where_clause_entity:       OPTIONAL where_clause;
END_ENTITY;
```

```
(* K Not in rules *)
ENTITY entity_block_wrapper
SUBTYPE OF (named_types);
 wrapped_entity_block:        entity_block;
END_ENTITY;

(* K 188, 190 *)
ENTITY enumeration_type
SUBTYPE OF (type_type);
 items:                       LIST [1:?] OF simple_id;
END_ENTITY;

(* K 192 *)
ENTITY explicit_attr;
 attribute_decls:             LIST [1:?] OF attribute_decl;
 is_optional:                 OPTIONAL BOOLEAN;
 base_type_entity:            base_type;
END_ENTITY;

(* 196 stub *)
ENTITY function_block
SUBTYPE OF (block);
END_ENTITY;

(* 216 stub *)
ENTITY inverse_clause;
END_ENTITY;

(* 219 *)
ENTITY labelled_expression;
 label_entity:                OPTIONAL label;
 expression_entity:           expression;
END_ENTITY;

(* K 184, 187, 226, 282, 284
It would have been nice to have this be a subtype of both block and base_type, but Fed-X-Plus
couldn't handle it (compiler errors). It has been done the following way as a work-around, also
avoiding SELECT.

*)
ENTITY named_types
ABSTRACT SUPERTYPE OF (ONEOF (entity_block_wrapper, type_decl_wrapper))
SUBTYPE OF (base_type);
END_ENTITY;
```

```
(* K 229 *)
ENTITY one_of
SUBTYPE OF (supertype_factor);
 supertype_expressions:        LIST[1:?] OF supertype_expression;
END_ENTITY;

(* 235 stub *)
ENTITY procedure_block
SUBTYPE OF (block);
END_ENTITY;

(* 245 stub *)
ENTITY reference_clause;
END_ENTITY;

(* 256 stub *)
ENTITY rule_block
SUBTYPE OF (block);
END_ENTITY;

(* K 176, 211, 259, 260 *)
ENTITY schema_block;
 schema_id:                    simple_id;
 reference_clauses:            LIST [0:?] OF reference_clause;
 use_clauses:                  LIST [0:?] OF use_clause;
 constant_decl_entity:         OPTIONAL constant_decl;
 blocks:                       LIST [0:?] OF block;
END_ENTITY;

(* 263 *)
ENTITY select_type
SUBTYPE OF (type_type);
 choices:                      LIST [1:?] OF named_types;
END_ENTITY;

(* K 268 *)
ENTITY simple_types
SUBTYPE OF (base_type);
 type_name:                    simple_type_name;
END_ENTITY;

(* Not in rules *)
ENTITY supertype_and
SUBTYPE OF (supertype_continuation);
 supertype_factor_entity:      supertype_factor;
END_ENTITY;
```

```
(* Not in rules *)
ENTITY supertype_andor
SUBTYPE OF (supertype_continuation);
 supertype_factor_entity:       supertype_factor;
END_ENTITY;

(* Not in rules *)
ENTITY supertype_continuation
ABSTRACT SUPERTYPE OF (ONEOF (supertype_and, supertype_andor));
END_ENTITY;

(* K 276 *)
ENTITY supertype_declaration;
 is_abstract:                   OPTIONAL BOOLEAN;
 supertype_expression_entity:OPTIONAL supertype_expression;
END_ENTITY;

(* K Not in rules *)
ENTITY supertype_entity_ref
SUBTYPE OF (supertype_factor);
 referenced_type:               entity_block;
END_ENTITY;

(* K 277 *)
ENTITY supertype_expression
SUBTYPE OF (supertype_factor);
 supertype_factor_entity:       supertype_factor;
 supertype_continuations:       LIST [0:?] OF supertype_continuation;
END_ENTITY;

(* K 278 *)
ENTITY supertype_factor
ABSTRACT SUPERTYPE OF (ONEOF (supertype_entity_ref, one_of, supertype_expression));
END_ENTITY;

(* K 281 *)
ENTITY type_decl
SUBTYPE OF (block);
 type_id:                       simple_id;
 underlying_type_entity:        underlying_type;
 where_clause_entity:           OPTIONAL where_clause;
(* WHERE the underlying_type is not an entity_block_wrapper *)
END_ENTITY;

(* K Not in rules *)
ENTITY type_decl_wrapper
SUBTYPE OF (named_types);
 wrapped_type_decl:             type_decl;
END_ENTITY;
```

```
(* Not in rules *)
ENTITY type_type
ABSTRACT SUPERTYPE OF (ONEOF (enumeration_type, select_type))
SUBTYPE OF (underlying_type);
END_ENTITY;

(* K 286 *)
ENTITY underlying_type
ABSTRACT SUPERTYPE OF (ONEOF (base_type, type_type));
END_ENTITY;

(* 287 stub *)
ENTITY unique_clause;
END_ENTITY;

(* 289 stub *)
ENTITY use_clause;
END_ENTITY;

(* 292 *)
ENTITY where_clause;
 labelled_expressions:          LIST [1:?] OF labelled_expression;
END_ENTITY;

END_SCHEMA;
```

D Algorithmically Derived EXPRESS Schema

This is an EXPRESS file called “Algorithmic EXPRESS” for a subset of the EXPRESS language derived by applying algorithms for generating EXPRESS statement from productions to the production language definition of EXPRESS. Appendices C, E, and F define the same subset of EXPRESS as does this appendix. This schema is discussed in section 5.2. The number of the production rule from which each EXPRESS statement was derived is given above the statement.

```
SCHEMA sexe_express;

(* 144 *)
TYPE aggregation_types = SELECT (array_type, bag_type, list_type, set_type);
END_TYPE;

(* 151 simplified *)
TYPE attribute_decl = simple_id;
END_TYPE;

(* 152 *)
TYPE attribute_id = simple_id;
END_TYPE;

 (* 156 *)
TYPE base_type = SELECT (aggregation_types, simple_types, named_types);
END_TYPE;

(* 157 simplified *)
TYPE binary_type = token_BINARY;
END_TYPE;

(* 158 *)
TYPE boolean_type = token_BOOLEAN;
END_TYPE;

(* 160 *)
TYPE bound_1 = simple_expression;
END_TYPE;

(* 161 *)
TYPE bound_2 = simple_expression;
END_TYPE;

(* 176 *)
TYPE declaration = SELECT (entity_block, function_block, procedure_block, type_decl);
END_TYPE;

(* 184 *)
TYPE entity_id = simple_id;
END_TYPE;
```

```
(* 187 *)
TYPE entity_ref = entity_id;
END_TYPE;

(* 188 *)
TYPE enumeration_id = simple_id;
END_TYPE;

(* 193 simplified *)
TYPE expression = STRING;
END_TYPE;

(* 210 *)
TYPE integer_type = token_INTEGER;
END_TYPE;

(* 211 *)
TYPE interface_specification = SELECT (reference_clause, use_clause);
END_TYPE;

(* 217 *)
TYPE label = simple_id;
END_TYPE;

(* 225 *)
TYPE logical_type = token_LOGICAL;
END_TYPE;

(* 226 *)
TYPE named_types = SELECT (entity_ref, type_ref);
END_TYPE;

(* 228 *)
TYPE number_type = token_NUMBER;
END_TYPE;

(* 244 simplified *)
TYPE real_type = token_REAL;
END_TYPE;

(* 261 *)
TYPE schema_id = simple_id;
END_TYPE;

(* 260 *)
TYPE select_1 = SELECT (declaration, rule_block);
END_TYPE;

(* 277 *)
TYPE select_2 = SELECT (token_AND, token_ANDOR);
END_TYPE;
```

```
(* 266 simplified *)
TYPE simple_expression = STRING;
END_TYPE;

(* 136 simplified *)
TYPE simple_id = STRING;
END_TYPE;

(* 268 simplified *)
TYPE simple_types = SELECT (binary_type, boolean_type, integer_type,
 logical_type, number_type, real_type, string_type2);
END_TYPE;

(* 271 simplified *)
(* Also, the name is changed so it is not multiply defined. *)
TYPE string_type2 = token_STRING;
END_TYPE;

(* 276 *)
TYPE supertype_declaration = SELECT (construct_5, construct_6);
END_TYPE;

(* 278 *)
TYPE supertype_factor = SELECT (entity_ref, one_of, construct_8);
END_TYPE;

(* 1 *)
TYPE token_ABSTRACT = ENUMERATION OF (string_abstract);
END_TYPE;

(* 5 *)
TYPE token_AND = ENUMERATION OF (string_and);
END_TYPE;

(* 6 *)
TYPE token_ANDOR = ENUMERATION OF (string_andor);
END_TYPE;

(* 7 *)
TYPE token_ARRAY = ENUMERATION OF (string_array);
END_TYPE;

(* No rule *)
TYPE token_colon = ENUMERATION OF (colon);
END_TYPE;

(* No rule *)
TYPE token_comma = ENUMERATION OF (comma);
END_TYPE;
```

```
(* 11 *)
TYPE token_BAG = ENUMERATION OF (string_bag);
END_TYPE;

(* 13 *)
TYPE token_BINARY = ENUMERATION OF (string_binary);
END_TYPE;

(* 15 *)
TYPE token_BOOLEAN = ENUMERATION OF (string_boolean);
END_TYPE;

(* 30 *)
TYPE token_END_ENTITY = ENUMERATION OF (string_end_entity);
END_TYPE;

(* 38 *)
TYPE token_END_SCHEMA = ENUMERATION OF (string_end_schema);
END_TYPE;

(* 39 *)
TYPE token_END_TYPE = ENUMERATION OF (string_end_type);
END_TYPE;

(* 40 *)
TYPE token_ENTITY = ENUMERATION OF (string_entity);
END_TYPE;

(* 41 *)
TYPE token_ENUMERATION = ENUMERATION OF (string_enumeration);
END_TYPE;

(* No rule *)
TYPE token_equals = ENUMERATION OF (equals);
END_TYPE;

(* 57 *)
TYPE token_INTEGER = ENUMERATION OF (string_integer);
END_TYPE;

(* No rule *)
TYPE token_left_brace = ENUMERATION OF (left_brace);
END_TYPE;

(* No rule *)
TYPE token_left_bracket = ENUMERATION OF (left_bracket);
END_TYPE;

(* No rule *)
TYPE token_left_parenthesis = ENUMERATION OF (left_parenthesis);
END_TYPE;
```

```
(* 61 *)
TYPE token_LIST = ENUMERATION OF (string_list);
END_TYPE;

(* 68 *)
TYPE token_LOGICAL = ENUMERATION OF (string_logical);
END_TYPE;

(* 72 *)
TYPE token_NUMBER = ENUMERATION OF (string_number);
END_TYPE;

(* 75 *)
TYPE token_OF = ENUMERATION OF (string_of);
END_TYPE;

(* 76 *)
TYPE token_ONEOF = ENUMERATION OF (string_oneof);
END_TYPE;

(* 77 *)
TYPE token_OPTIONAL = ENUMERATION OF (string_optional);
END_TYPE;

(* 83 *)
TYPE token_REAL = ENUMERATION OF (string_real);
END_TYPE;

(* No rule *)
TYPE token_right_brace = ENUMERATION OF (right_brace);
END_TYPE;

(* No rule *)
TYPE token_right_bracket = ENUMERATION OF (right_bracket);
END_TYPE;

(* No rule *)
TYPE token_right_parenthesis = ENUMERATION OF (right_parenthesis);
END_TYPE;

(* 90 *)
TYPE token_SCHEMA = ENUMERATION OF (string_schema);
END_TYPE;

(* 91 *)
TYPE token_SELECT = ENUMERATION OF (string_select);
END_TYPE;

(* No rule *)
TYPE token_semicolon = ENUMERATION OF (semicolon);
END_TYPE;
```

```
(* 93 *)
TYPE token_SET = ENUMERATION OF (string_set);
END_TYPE;

(* 98 *)
TYPE token_STRING = ENUMERATION OF (string_string);
END_TYPE;

(* 99 *)
TYPE token_SUBTYPE = ENUMERATION OF (string_subtype);
END_TYPE;

(* 100 *)
TYPE token_SUPERTYPE = ENUMERATION OF (string_supertype);
END_TYPE;

(* 105 *)
TYPE token_TYPE = ENUMERATION OF (string_type);
END_TYPE;

(* 107 *)
TYPE token_UNIQUE = ENUMERATION OF (string_unique);
END_TYPE;

(* 114 *)
TYPE token_WHERE = ENUMERATION OF (string_where);
END_TYPE;

(* 282 *)
TYPE type_id = simple_id;
END_TYPE;

(* 284 *)
TYPE type_ref = type_id;
END_TYPE;

(* 286 *)
TYPE underlying_type = SELECT (enumeration_type, select_type,
 aggregation_types, simple_types, type_ref);
END_TYPE;

(* 149 corrected from Appendix B *)
ENTITY array_type;
 token_ARRAY_at:        token_ARRAY;
 bound_spec_at:         bound_spec;
 token_OF_at:           token_OF;
 token_OPTIONAL_at:     OPTIONAL token_OPTIONAL;
 token_UNIQUE_at:       OPTIONAL token_UNIQUE;
 base_type_at:          base_type;
END_ENTITY;
```

```
(* 155 *)
ENTITY bag_type;
 token_BAG_at:              token_BAG;
 bound_spec_at:             OPTIONAL bound_spec;
 token_OF_at:               token_OF;
 base_type_at:              base_type;
END_ENTITY;

(* 159 *)
ENTITY bound_spec;
 left_parenthesis_at:       token_left_parenthesis;
 bound_1_at:                bound_1;
 token_colon_at:            token_colon;
 bound_2_at:                bound_2;
 right_parenthesis_at:      token_right_parenthesis;
END_ENTITY;

(* 171 stub *)
ENTITY constant_decl;
END_ENTITY;

(* 192 *)
ENTITY construct_1;
 token_comma_at:            token_comma;
 attribute_decl_at:         attribute_decl;
END_ENTITY;

(* 219 *)
ENTITY construct_2;
 label_at:                  label;
 token_colon_at:            token_colon;
END_ENTITY;

(* 229 *)
ENTITY construct_3;
 token_comma_at:            token_comma;
 supertype_expression_at:   supertype_expression;
END_ENTITY;

(* 275 *)
ENTITY construct_4;
 token_comma_at:            token_comma;
 entity_ref_at:             entity_ref;
END_ENTITY;
```

```
(* 276 *)
ENTITY construct_5;
 token_ABSTRACT_at:       token_ABSTRACT;
 token_SUPERTYPE_at:      token_SUPERTYPE;
END_ENTITY;

(* 276 *)
ENTITY construct_6;
 token_ABSTRACT_at:       OPTIONAL token_ABSTRACT;
 token_SUPERTYPE_at:      token_SUPERTYPE;
 token_OF_at:             token_OF;
 token_left_parenthesis_at:  token_left_parenthesis;
 supertype_expression_at:    supertype_expression;
 token_right_parenthesis_at: token_right_parenthesis;
END_ENTITY;

(* 277 *)
ENTITY construct_7;
 select_2_at:select_2;
 supertype_factor_at:        supertype_factor;
END_ENTITY;

(* 278 *)
ENTITY construct_8;
 token_left_parenthesis_at:  token_left_parenthesis;
 supertype_expression_at:    supertype_expression;
 token_right_parenthesis_at: token_right_parenthesis;
END_ENTITY;

(* 292 *)
ENTITY construct_9;
 labelled_expression_at:  labelled_expression;
 token_semicolon_at:      token_semicolon;
END_ENTITY;

(* 190 *)
ENTITY construct_10;
 token_comma_at:          token_comma;
 enumeration_id_at:       enumeration_id;
END_ENTITY;

(* 263 *)
ENTITY construct_11;
 token_comma_at:          token_comma;
 named_types_at:          named_types;
END_ENTITY;
```

```
(* 178 stub *)
ENTITY derive_clause;
END_ENTITY;

(* 181 *)
ENTITY entity_block;
 entity_head_at:              entity_head;
 entity_body_at:              entity_body;
 token_END_ENTITY_at:         token_END_ENTITY;
 token_semicolon_at:          token_semicolon;
END_ENTITY;

(* 182 *)
ENTITY entity_body;
 explicit_attrs:              LIST [0:?] OF explicit_attr;
 derive_clause_at:            OPTIONAL derive_clause;
 inverse_clause_at:           OPTIONAL inverse_clause;
 unique_clause_at:            OPTIONAL unique_clause;
 where_clause_at:             OPTIONAL where_clause;
END_ENTITY;

(* 183 *)
ENTITY entity_head;
 token_ENTITY_at:             token_ENTITY;
 entity_id_at:                entity_id;
 subsuper_at:                 OPTIONAL subsuper;
 token_semicolon_at:          token_semicolon;
END_ENTITY;

(* 190 *)
ENTITY enumeration_type;
 token_ENUMERATION_at:        token_ENUMERATION;
 token_OF_at:                 token_OF;
 token_left_parenthesis_at:   token_left_parenthesis;
 enumeration_id_at:           enumeration_id;
 construct_10s:               LIST [0:?] OF construct_10;
 token_right_parenthesis_at:  token_right_parenthesis;
END_ENTITY;

(* 192 *)
ENTITY explicit_attr;
 attribute_decl_at:           attribute_decl;
 construct_1s:                LIST [0:?] OF construct_1;
 token_colon_at:              token_colon;
 token_OPTIONAL_at:           OPTIONAL token_OPTIONAL;
 base_type_at:                base_type;
 token_semicolon_at:          token_semicolon;
END_ENTITY;
```

```
(* 196 stub *)
ENTITY function_block;
END_ENTITY;

(* 216 stub *)
ENTITY inverse_clause;
END_ENTITY;

(* 219 *)
ENTITY labelled_expression;
 construct_2_at:        OPTIONAL construct_2;
 expression_at:         expression;
END_ENTITY;

(* 220 *)
ENTITY list_type;
 token_LIST_at:         token_LIST;
 bound_spec_at:         OPTIONAL bound_spec;
 token_OF_at:           token_OF;
 token_UNIQUE_at:       OPTIONAL token_UNIQUE;
 base_type_at:          base_type;
END_ENTITY;

(* 229 *)
ENTITY one_of;
 token_ONEOF_at:            token_ONEOF;
 token_left_parenthesis_at: token_left_parenthesis;
 supertype_expression_at:   supertype_expression;
 construct_3s:              LIST [0:?] OF construct_3;
 token_right_parenthesis_at: token_right_parenthesis;
END_ENTITY;

(* 235 stub *)
ENTITY procedure_block;
END_ENTITY;

(* 245 stub *)
ENTITY reference_clause;
END_ENTITY;

(* 256 stub *)
ENTITY rule_block;
END_ENTITY;
```

```
(* 259 *)
ENTITY schema_block;
 token_SCHEMA_at:          token_SCHEMA;
 schema_id_at:             schema_id;
 token_semicolon_at:       token_semicolon;
 schema_body_at:           schema_body;
 token_END_SCHEMA_at:      token_END_SCHEMA;
 token_semicolon_at2:      token_semicolon;
END_ENTITY;

(* 260 *)
ENTITY schema_body;
 interface_specifications: LIST [0:?] OF interface_specification;
 constant_decl_at:         OPTIONAL constant_decl;
 select_1s:                LIST [0:?] OF select_1;
END_ENTITY;

(* 263 *)
ENTITY select_type;
 token_SELECT_at:          token_SELECT;
 token_left_parenthesis_at: token_left_parenthesis;
 named_types_at:           named_types;
 construct_11s:            LIST [0:?] OF construct_11;
 token_right_parenthesis_at: token_right_parenthesis;
END_ENTITY;

(* 265 *)
ENTITY set_type;
 token_SET_at:             token_SET;
 bound_spec_at:            OPTIONAL bound_spec;
 token_OF_at:              token_OF;
 base_type_at:             base_type;
END_ENTITY;

(* 273 *)
ENTITY subsuper;
 supertype_declaration_at: OPTIONAL supertype_declaration;
 subtype_declaration_at:   OPTIONAL subtype_declaration;
END_ENTITY;
```

```
(* 275 *)
ENTITY subtype_declaration;
 token_SUBTYPE_at:          token_SUBTYPE;
 token_OF_at:               token_OF;
 token_left_parenthesis_at: token_left_parenthesis;
 entity_ref_at:             entity_ref;
 construct_4s:              LIST [0:?] OF construct_4;
 token_right_parenthesis_at: token_right_parenthesis;
END_ENTITY;

(* 277 *)
ENTITY supertype_expression;
 supertype_factor_at:       supertype_factor;
 construct_7s:              LIST [0:?] OF construct_7;
END_ENTITY;

(* 281 *)
ENTITY type_decl;
 token_TYPE_at:             token_TYPE;
 type_id_at:                type_id;
 token_equals_at:           token_equals;
 underlying_type_at:        underlying_type;
 token_semicolon_at:        token_semicolon;
 where_clause_at:           OPTIONAL where_clause;
 token_END_TYPE_at:         token_END_TYPE;
 token_semicolon_at2:       token_semicolon;
END_ENTITY;

(* 287 stub *)
ENTITY unique_clause;
END_ENTITY;

(* 289 stub *)
ENTITY use_clause;
END_ENTITY;

(* 292 *)
ENTITY where_clause;
 token_WHERE_at:            token_WHERE;
 labelled_expression_at:    labelled_expression;
 token_semicolon_at:        token_semicolon;
 construct_9s:              LIST [0:?] OF construct_9;
END_ENTITY;

END_SCHEMA;
```

E STEP Exchange File for EXPRESS

This is an exchange file called “Exchange File EXPRESS” for an EXPRESS schema for a subset of the EXPRESS language. The file was created by one of the authors using the SEXE Interactive Editor to replicate the schema given in Appendix C. Appendices C, D, and F define the same subset of EXPRESS as does this appendix. This schema is discussed in section 5.3.

```
STEP;
HEADER;
ENDSEC;
DATA;
@1=SCHEMA_BLOCK('sexe_express',,,,(#2,#4,#6,#7,#8,#9,#10,#12,#13,#14,#15,#16,#17,
#18,#19,#20,#21,#22,#23,#24,#25,#26,#27,#28,#29,#30,#31,#32,#33,#34,#35,#36,#37,#38,#39,
#40,#41,#42,#43,#44,#45,#46));
@2=TYPE_DECL('aggregation_name',#3,);
@3=ENUMERATION_TYPE(('string_array', 'string_bag', 'string_list', 'string_set'));
@4=TYPE_DECL('attribute_decl',#5,);
@5=SIMPLE_TYPES(.STRING_STRING.);
@6=TYPE_DECL('expression',#5,);
@7=TYPE_DECL('label',#5,);
@8=TYPE_DECL('simple_expression',#5,);
@9=TYPE_DECL('simple_id',#5,);
@10=TYPE_DECL('simple_type_name',#11,);
@11=ENUMERATION_TYPE(('string_binary', 'string_boolean', 'string_integer',
'string_logical', 'string_number', 'string_real', 'string_string'));
@12=ENTITY_BLOCK('aggregation_types',,(#13),(#47,#48,#49,#50,#51),,,,);
@13=ENTITY_BLOCK('base_type',#89,(#43),,,,,);
@14=ENTITY_BLOCK('block',#110,,,,,,);
@15=ENTITY_BLOCK('bound_spec',,,(#52,#53),,,,);
@16=ENTITY_BLOCK('constant_decl',,,,,,,);
@17=ENTITY_BLOCK('derive_clause',,,,,,,);
@18=ENTITY_BLOCK('entity_block',,(#14),(#54,#55,#56,#57,#58,#59,#60,#61),,,,);
@19=ENTITY_BLOCK('entity_block_wrapper',,(#25),(#62),,,,);
@20=ENTITY_BLOCK('enumeration_type',,(#42),(#63),,,,);
@21=ENTITY_BLOCK('explicit_attr',,,(#64,#65,#66),,,,);
@22=ENTITY_BLOCK('function_block',,(#14),,,,,);
@23=ENTITY_BLOCK('inverse_clause',,,,,,,);
@24=ENTITY_BLOCK('labelled_expression',,,(#67,#68),,,,);
@25=ENTITY_BLOCK('named_types',#117,(#13),,,,,);
@26=ENTITY_BLOCK('one_of',,(#39),(#69),,,,);
@27=ENTITY_BLOCK('procedure_block',,(#14),,,,,);
@28=ENTITY_BLOCK('reference_clause',,,,,,,);
@29=ENTITY_BLOCK('rule_block',,(#14),,,,,);
@30=ENTITY_BLOCK('schema_block',,,(#70,#71,#72,#73,#74),,,,);
@31=ENTITY_BLOCK('select_type',,(#42),(#75),,,,);
@32=ENTITY_BLOCK('simple_types',,(#13),(#76),,,,);
```

@33=ENTITY_BLOCK('supertype_and',,(#35),(#77),,,,);
@34=ENTITY_BLOCK('supertype_andor',,(#35),(#78),,,,);
@35=ENTITY_BLOCK('supertype_continuation',#124,,,,,,);
@36=ENTITY_BLOCK('supertype_declaration',,,(#79,#80),,,,);
@37=ENTITY_BLOCK('supertype_entity_ref',,(#39),(#81),,,,);
@38=ENTITY_BLOCK('supertype_expression',,(#39),(#82,#83),,,,);
@39=ENTITY_BLOCK('supertype_factor',#133,,,,,,);
@40=ENTITY_BLOCK('type_decl',,(#14),(#84,#85,#86),,,,);
@41=ENTITY_BLOCK('type_decl_wrapper',,(#25),(#87),,,,);
@42=ENTITY_BLOCK('type_type',#140,(#43),,,,,);
@43=ENTITY_BLOCK('underlying_type',#147,,,,,,);
@44=ENTITY_BLOCK('unique_clause',,,,,,,);
@45=ENTITY_BLOCK('use_clause',,,,,,,);
@46=ENTITY_BLOCK('where_clause',,,(#88),,,,);
@47=EXPLICIT_ATTR(('aggregation_type_name'),,#148);
@48=EXPLICIT_ATTR(('bound_spec_entity'),.T.,#149);
@49=EXPLICIT_ATTR(('is_optional'),.T.,#150);
@50=EXPLICIT_ATTR(('is_unique'),.T.,#150);
@51=EXPLICIT_ATTR(('base_type_entity'),,#151);
@52=EXPLICIT_ATTR(('bound_1'),,#152);
@53=EXPLICIT_ATTR(('bound_2'),,#152);
@54=EXPLICIT_ATTR(('entity_id'),,#153);
@55=EXPLICIT_ATTR(('supertype_declaration_entity'),.T.,#157);
@56=EXPLICIT_ATTR(('subtype_declaration'),.T.,#154);
@57=EXPLICIT_ATTR(('explicit_attrs'),,#158);
@58=EXPLICIT_ATTR(('derive_clause_entity'),.T.,#161);
@59=EXPLICIT_ATTR(('inverse_clause_entity'),.T.,#162);
@60=EXPLICIT_ATTR(('unique_clause_entity'),.T.,#164);
@61=EXPLICIT_ATTR(('where_clause_entity'),.T.,#163);
@62=EXPLICIT_ATTR(('wrapped_entity_block'),,#156);
@63=EXPLICIT_ATTR(('items'),,#165);
@64=EXPLICIT_ATTR(('attribute_decls'),,#167);
@65=EXPLICIT_ATTR(('is_optional'),.T.,#150);
@66=EXPLICIT_ATTR(('base_type_entity'),,#151);
@67=EXPLICIT_ATTR(('label_entity'),.T.,#170);
@68=EXPLICIT_ATTR(('expression_entity'),,#171);
@69=EXPLICIT_ATTR(('supertype_expressions'),,#172);
@70=EXPLICIT_ATTR(('schema_id'),,#153);
@71=EXPLICIT_ATTR(('reference_clauses'),,#175);
@72=EXPLICIT_ATTR(('use_clauses'),,#178);
@73=EXPLICIT_ATTR(('constant_decl_entity'),.T.,#181);
@74=EXPLICIT_ATTR(('blocks'),,#182);
@75=EXPLICIT_ATTR(('choices'),,#185);
@76=EXPLICIT_ATTR(('type_name'),,#188);
@77=EXPLICIT_ATTR(('supertype_factor_entity'),,#189);
@78=EXPLICIT_ATTR(('supertype_factor_entity'),,#189);

```
@79=EXPLICIT_ATTR((‘is_abstract’),.T.,#150);
@80=EXPLICIT_ATTR((‘supertype_expression_entity’),.T.,#174);
@81=EXPLICIT_ATTR((‘referenced_type’),,#156);
@82=EXPLICIT_ATTR((‘supertype_factor_entity’),,#189);
@83=EXPLICIT_ATTR((‘supertype_continuations’),,#190);
@84=EXPLICIT_ATTR((‘type_id’),,#153);
@85=EXPLICIT_ATTR((‘underlying_type_entity’),,#193);
@86=EXPLICIT_ATTR((‘where_clause_entity’),.T.,#163);
@87=EXPLICIT_ATTR((‘wrapped_type_decl’),,#194);
@88=EXPLICIT_ATTR((‘labelled_expressions’),,#195);
@89=SUPERTYPE_DECLARATION(.T.,#90);
@90=SUPERTYPE_EXPRESSION(#91,);
@91=ONE_OF((#95,#97,#98));
@92=SUPERTYPE_ENTITY_REF(#12);
@93=SUPERTYPE_ENTITY_REF(#25);
@94=SUPERTYPE_ENTITY_REF(#32);
@95=SUPERTYPE_EXPRESSION(#92,);
@96=SUPERTYPE_ENTITY_REF(#18);
@97=SUPERTYPE_EXPRESSION(#93,);
@98=SUPERTYPE_EXPRESSION(#94,);
@99=SUPERTYPE_ENTITY_REF(#29);
@100=SUPERTYPE_ENTITY_REF(#40);
@101=SUPERTYPE_ENTITY_REF(#27);
@102=SUPERTYPE_ENTITY_REF(#22);
@103=SUPERTYPE_EXPRESSION(#96,);
@104=SUPERTYPE_EXPRESSION(#99,);
@105=SUPERTYPE_EXPRESSION(#100,);
@106=SUPERTYPE_EXPRESSION(#101,);
@107=SUPERTYPE_EXPRESSION(#102,);
@108=ONE_OF((#103,#104,#105,#106,#107));
@109=SUPERTYPE_EXPRESSION(#108,);
@110=SUPERTYPE_DECLARATION(.T.,#109);
@111=SUPERTYPE_ENTITY_REF(#19);
@112=SUPERTYPE_ENTITY_REF(#41);
@113=SUPERTYPE_EXPRESSION(#111,);
@114=SUPERTYPE_EXPRESSION(#112,);
@115=ONE_OF((#113,#114));
@116=SUPERTYPE_EXPRESSION(#115,);
@117=SUPERTYPE_DECLARATION(.T.,#116);
@118=SUPERTYPE_ENTITY_REF(#33);
@119=SUPERTYPE_EXPRESSION(#118,);
@120=SUPERTYPE_ENTITY_REF(#34);
@121=SUPERTYPE_EXPRESSION(#120,);
@122=ONE_OF((#119,#121));
@123=SUPERTYPE_EXPRESSION(#122,);
@124=SUPERTYPE_DECLARATION(.T.,#123);
```

```
@125=SUPERTYPE_ENTITY_REF(#37);
@126=SUPERTYPE_ENTITY_REF(#26);
@127=SUPERTYPE_ENTITY_REF(#38);
@128=SUPERTYPE_EXPRESSION(#125,);
@129=SUPERTYPE_EXPRESSION(#126,);
@130=SUPERTYPE_EXPRESSION(#127,);
@131=ONE_OF((#128,#129,#130));
@132=SUPERTYPE_EXPRESSION(#131,);
@133=SUPERTYPE_DECLARATION(.T.,#132);
@134=SUPERTYPE_ENTITY_REF(#20);
@135=SUPERTYPE_ENTITY_REF(#31);
@136=SUPERTYPE_EXPRESSION(#134,);
@137=SUPERTYPE_EXPRESSION(#135,);
@138=ONE_OF((#136,#137));
@139=SUPERTYPE_EXPRESSION(#138,);
@140=SUPERTYPE_DECLARATION(.T.,#139);
@141=SUPERTYPE_ENTITY_REF(#13);
@142=SUPERTYPE_ENTITY_REF(#42);
@143=SUPERTYPE_EXPRESSION(#141,);
@144=SUPERTYPE_EXPRESSION(#142,);
@145=ONE_OF((#143,#144));
@146=SUPERTYPE_EXPRESSION(#145,);
@147=SUPERTYPE_DECLARATION(.T.,#146);
@148=TYPE_DECL_WRAPPER(#2);
@149=ENTITY_BLOCK_WRAPPER(#15);
@150=SIMPLE_TYPES(.STRING_BOOLEAN.);
@151=ENTITY_BLOCK_WRAPPER(#13);
@152=TYPE_DECL_WRAPPER(#8);
@153=TYPE_DECL_WRAPPER(#9);
@154=AGGREGATION_TYPES(.STRING_LIST.,#155,,,#156);
@155=BOUND_SPEC('1','?');
@156=ENTITY_BLOCK_WRAPPER(#18);
@157=ENTITY_BLOCK_WRAPPER(#36);
@158=AGGREGATION_TYPES(.STRING_LIST.,#159,,,#160);
@159=BOUND_SPEC('0','?');
@160=ENTITY_BLOCK_WRAPPER(#21);
@161=ENTITY_BLOCK_WRAPPER(#17);
@162=ENTITY_BLOCK_WRAPPER(#23);
@163=ENTITY_BLOCK_WRAPPER(#46);
@164=ENTITY_BLOCK_WRAPPER(#44);
@165=AGGREGATION_TYPES(.STRING_LIST.,#166,,,#153);
@166=BOUND_SPEC('1','?');
@167=AGGREGATION_TYPES(.STRING_LIST.,#168,,,#169);
@168=BOUND_SPEC('1','?');
@169=TYPE_DECL_WRAPPER(#4);
@170=TYPE_DECL_WRAPPER(#7);
```

```
@171=TYPE_DECL_WRAPPER(#6);
@172=AGGREGATION_TYPES(.STRING_
LIST.,#173,,,#174);
@173=BOUND_SPEC('1','?');
@174=ENTITY_BLOCK_WRAPPER(#38);
@175=AGGREGATION_TYPES(.STRING_
LIST.,#176,,,#177);
@176=BOUND_SPEC('0','?');
@177=ENTITY_BLOCK_WRAPPER(#28);
@178=AGGREGATION_TYPES(.STRING_
LIST.,#179,,,#180);
@179=BOUND_SPEC('0','?');
@180=ENTITY_BLOCK_WRAPPER(#45);
@181=ENTITY_BLOCK_WRAPPER(#16);
@182=AGGREGATION_TYPES(.STRING_
LIST.,#183,,,#184);
@183=BOUND_SPEC('0','?');
@184=ENTITY_BLOCK_WRAPPER(#14);
@185=AGGREGATION_TYPES(.STRING_
LIST.,#186,,,#187);
@186=BOUND_SPEC('1','?');
@187=ENTITY_BLOCK_WRAPPER(#25);
@188=TYPE_DECL_WRAPPER(#10);
@189=ENTITY_BLOCK_WRAPPER(#39);
@190=AGGREGATION_TYPES(.STRING_
LIST.,#191,,,#192);
@191=BOUND_SPEC('0','?');
@192=ENTITY_BLOCK_WRAPPER(#35);
@193=ENTITY_BLOCK_WRAPPER(#43);
@194=ENTITY_BLOCK_WRAPPER(#40);
@195=AGGREGATION_TYPES(.STRING_
LIST.,#196,,,#197);
@196=BOUND_SPEC('1','?');
@197=ENTITY_BLOCK_WRAPPER(#24);
ENDSEC;
ENDSTEP;
```

F LISP Property List for EXPRESS

This is a LISP-readable file called “LISP EXPRESS” describing an EXPRESS schema for a subset of the EXPRESS language. The file was created by running the file of Appendix E through the “step-to-plist” and “pp-plist” LISP utilities. Appendices C, D, and E define the same subset of EXPRESS as does this appendix. This schema is discussed in section 5.4.

```
(setf (symbol-plist ‘express) ‘
(header (header
   file_identification (file_identification
   file_name nil
   date nil
   author nil
   organization nil
   step_version nil
   preprocessor_version nil
   originating_system nil)
   file_description nil
   imp_level nil)
data (data

1 (1 type schema_block
   schema_id “sexe_express”
   reference_clauses nil
   use_clauses nil
   constant_decl_entity nil
   blocks ((ref 2) (ref 4) (ref 6) (ref 7)
      (ref 8) (ref 9) (ref 10) (ref 12)
      (ref 13) (ref 14) (ref 15) (ref 16)
      (ref 17) (ref 18) (ref 19) (ref 20)
      (ref 21) (ref 22) (ref 23) (ref 24)
      (ref 25) (ref 26) (ref 27) (ref 28)
      (ref 29) (ref 30) (ref 31) (ref 32)
      (ref 33) (ref 34) (ref 35) (ref 36)
      (ref 37) (ref 38) (ref 39) (ref 40)
      (ref 41) (ref 42) (ref 43) (ref 44)
      (ref 45) (ref 46)))

2 (2 type type_decl
   type_id “aggregation_name”
   underlying_type_entity (ref 3)
   where_clause_entity nil)

3 (3 type enumeration_type
   items (“string_array” “string_bag”
   “string_list” “string_set”))

4 (4 type type_decl
   type_id “attribute_decl”
   underlying_type_entity (ref 5)
   where_clause_entity nil)

5 (5 type simple_types
   type_name string_string)

6 (6 type type_decl
   type_id “expression”
   underlying_type_entity (ref 5)
   where_clause_entity nil)

7 (7 type type_decl
   type_id “label”
   underlying_type_entity (ref 5)
   where_clause_entity nil)

8 (8 type type_decl
   type_id “simple_expression”
   underlying_type_entity (ref 5)
   where_clause_entity nil)

9 (9 type type_decl
   type_id “simple_id”
   underlying_type_entity (ref 5)
   where_clause_entity nil)

10 (10 type type_decl
   type_id “simple_type_name”
   underlying_type_entity (ref 11)
   where_clause_entity nil)

11 (11 type enumeration_type
   items (“string_binary”
   “string_boolean” “string_integer”
   “string_logical” “string_number”
   “string_real” “string_string”))

12 (12 type entity_block
   entity_id “aggregation_types”
   supertype_declaration_entity nil
   subtype_declaration ((ref 13))
   explicit_attrs ((ref 47)
   (ref 48) (ref 49)
   (ref 50) (ref 51))
   derive_clause_entity nil
   inverse_clause_entity nil
   unique_clause_entity nil
   where_clause_entity nil)
```

13 (13 type entity_block
entity_id “base_type”
supertype_declaration_entity (ref 89)
subtype_declaration ((ref 43))
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

14 (14 type entity_block
entity_id “block”
supertype_declaration_entity (ref 110)
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

15 (15 type entity_block
entity_id “bound_spec”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs ((ref 52) (ref 53))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

16 (16 type entity_block
entity_id “constant_decl”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

17 (17 type entity_block
entity_id “derive_clause”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

18 (18 type entity_block
entity_id “entity_block”
supertype_declaration_entity nil
subtype_declaration ((ref 14))
explicit_attrs ((ref 54) (ref 55)
(ref 56) (ref 57)
(ref 58) (ref 59)
(ref 60) (ref 61))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

19 (19 type entity_block
entity_id “entity_block_wrapper”
supertype_declaration_entity nil
subtype_declaration ((ref 25))
explicit_attrs ((ref 62))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

20 (20 type entity_block
entity_id “enumeration_type”
supertype_declaration_entity nil
subtype_declaration ((ref 42))
explicit_attrs ((ref 63))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

21 (21 type entity_block
entity_id “explicit_attr”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs ((ref 64)
(ref 65) (ref 66))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

22 (22 type entity_block
entity_id “function_block”
supertype_declaration_entity nil
subtype_declaration ((ref 14))
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

23 (23 type entity_block
entity_id “inverse_clause”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

24 (24 type entity_block
entity_id “labelled_expression”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs ((ref 67) (ref 68))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

25 (25 type entity_block
entity_id “named_types”
supertype_declaration_entity (ref 117)
subtype_declaration ((ref 13))
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

26 (26 type entity_block
entity_id “one_of”
supertype_declaration_entity nil
subtype_declaration ((ref 39))
explicit_attrs ((ref 69))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

27 (27 type entity_block
entity_id “procedure_block”
supertype_declaration_entity nil
subtype_declaration ((ref 14))
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

28 (28 type entity_block
entity_id “reference_clause”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

29 (29 type entity_block
entity_id “rule_block”
supertype_declaration_entity nil
subtype_declaration ((ref 14))
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

30 (30 type entity_block
entity_id “schema_block”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs ((ref 70)
(ref 71) (ref 72)
(ref 73) (ref 74))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

31 (31 type entity_block
entity_id “select_type”
supertype_declaration_entity nil
subtype_declaration ((ref 42))
explicit_attrs ((ref 75))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

32 (32 type entity_block
entity_id “simple_types”
supertype_declaration_entity nil
subtype_declaration ((ref 13))
explicit_attrs ((ref 76))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

33 (33 type entity_block
entity_id “supertype_and”
supertype_declaration_entity nil
subtype_declaration ((ref 35))
explicit_attrs ((ref 77))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

34 (34 type entity_block
entity_id “supertype_andor”
supertype_declaration_entity nil
subtype_declaration ((ref 35))
explicit_attrs ((ref 78))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

35 (35 type entity_block
entity_id “supertype_continuation”
supertype_declaration_entity (ref 124)
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

36 (36 type entity_block
entity_id “supertype_declaration”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs ((ref 79) (ref 80))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

37 (37 type entity_block
entity_id “supertype_entity_ref”
supertype_declaration_entity nil
subtype_declaration ((ref 39))
explicit_attrs ((ref 81))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

38 (38 type entity_block
entity_id “supertype_expression”
supertype_declaration_entity nil
subtype_declaration ((ref 39))
explicit_attrs ((ref 82) (ref 83))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

39 (39 type entity_block
entity_id “supertype_factor”
supertype_declaration_entity (ref 133)
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

40 (40 type entity_block
entity_id “type_decl”
supertype_declaration_entity nil
subtype_declaration ((ref 14))
explicit_attrs ((ref 84)
(ref 85) (ref 86))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

41 (41 type entity_block
entity_id “type_decl_wrapper”
supertype_declaration_entity nil
subtype_declaration ((ref 25))
explicit_attrs ((ref 87))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

42 (42 type entity_block
entity_id “type_type”
supertype_declaration_entity (ref 140)
subtype_declaration ((ref 43))
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

43 (43 type entity_block
entity_id “underlying_type”
supertype_declaration_entity (ref 147)
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

44 (44 type entity_block
entity_id “unique_clause”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

45 (45 type entity_block
entity_id “use_clause”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs nil
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

46 (46 type entity_block
entity_id “where_clause”
supertype_declaration_entity nil
subtype_declaration nil
explicit_attrs ((ref 88))
derive_clause_entity nil
inverse_clause_entity nil
unique_clause_entity nil
where_clause_entity nil)

47 (47 type explicit_attr
attribute_decls (“aggregation_type_name”)
is_optional nil
base_type_entity (ref 148))

48 (48 type explicit_attr
attribute_decls (“bound_spec_entity”)
is_optional true
base_type_entity (ref 149))

49 (49 type explicit_attr
attribute_decls (“is_optional”)
is_optional true
base_type_entity (ref 150))

50 (50 type explicit_attr
attribute_decls (“is_unique”)
is_optional true
base_type_entity (ref 150))

51 (51 type explicit_attr
attribute_decls (“base_type_entity”)
is_optional nil
base_type_entity (ref 151))

52 (52 type explicit_attr
attribute_decls (“bound_1”)
is_optional nil
base_type_entity (ref 152))

53 (53 type explicit_attr
attribute_decls (“bound_2”)
is_optional nil
base_type_entity (ref 152))

54 (54 type explicit_attr
attribute_decls (“entity_id”)
is_optional nil
base_type_entity (ref 153))

55 (55 type explicit_attr
attribute_decls
(“supertype_declaration_entity”)
is_optional true
base_type_entity (ref 157))

56 (56 type explicit_attr
attribute_decls (“subtype_declaration”)
is_optional true
base_type_entity (ref 154))

57 (57 type explicit_attr
attribute_decls (“explicit_attrs”)
is_optional nil
base_type_entity (ref 158))

58 (58 type explicit_attr
attribute_decls (“derive_clause_entity”)
is_optional true
base_type_entity (ref 161))

59 (59 type explicit_attr
attribute_decls (“inverse_clause_entity”)
is_optional true
base_type_entity (ref 162))

60 (60 type explicit_attr
attribute_decls (“unique_clause_entity”)
is_optional true
base_type_entity (ref 164))

61 (61 type explicit_attr
attribute_decls (“where_clause_entity”)
is_optional true
base_type_entity (ref 163))

62 (62 type explicit_attr
attribute_decls (“wrapped_entity_block”)
is_optional nil
base_type_entity (ref 156))

63 (63 type explicit_attr
attribute_decls (“items”)
is_optional nil
base_type_entity (ref 165))

64 (64 type explicit_attr
attribute_decls (“attribute_decls”)
is_optional nil
base_type_entity (ref 167))

65 (65 type explicit_attr
attribute_decls (“is_optional”)
is_optional true
base_type_entity (ref 150))

66 (66 type explicit_attr
attribute_decls (“base_type_entity”)
is_optional nil
base_type_entity (ref 151))

67 (67 type explicit_attr
attribute_decls (“label_entity”)
is_optional true
base_type_entity (ref 170))

68 (68 type explicit_attr
attribute_decls (“expression_entity”)
is_optional nil
base_type_entity (ref 171))

69 (69 type explicit_attr
attribute_decls (“supertype_expressions”)
is_optional nil
base_type_entity (ref 172))

70 (70 type explicit_attr
attribute_decls (“schema_id”)
is_optional nil
base_type_entity (ref 153))

71 (71 type explicit_attr
attribute_decls (“reference_clauses”)
is_optional nil
base_type_entity (ref 175))

72 (72 type explicit_attr
attribute_decls (“use_clauses”)
is_optional nil
base_type_entity (ref 178))

73 (73 type explicit_attr
attribute_decls (“constant_decl_entity”)
is_optional true
base_type_entity (ref 181))

74 (74 type explicit_attr
attribute_decls (“blocks”)
is_optional nil
base_type_entity (ref 182))

75 (75 type explicit_attr
attribute_decls (“choices”)
is_optional nil
base_type_entity (ref 185))

76 (76 type explicit_attr
attribute_decls (“type_name”)
is_optional nil
base_type_entity (ref 188))

77 (77 type explicit_attr
attribute_decls (“supertype_factor_entity”)
is_optional nil
base_type_entity (ref 189))

78 (78 type explicit_attr
attribute_decls (“supertype_factor_entity”)
is_optional nil
base_type_entity (ref 189))

79 (79 type explicit_attr
attribute_decls (“is_abstract”)
is_optional true
base_type_entity (ref 150))

80 (80 type explicit_attr
attribute_decls
(“supertype_expression_entity”)
is_optional true
base_type_entity (ref 174))

81 (81 type explicit_attr
attribute_decls (“referenced_type”)
is_optional nil
base_type_entity (ref 156))

82 (82 type explicit_attr
attribute_decls (“supertype_factor_entity”)
is_optional nil
base_type_entity (ref 189))

83 (83 type explicit_attr
attribute_decls (“supertype_continuations”)
is_optional nil
base_type_entity (ref 190))

84 (84 type explicit_attr
attribute_decls (“type_id”)
is_optional nil
base_type_entity (ref 153))

85 (85 type explicit_attr
attribute_decls (“underlying_type_entity”)
is_optional nil
base_type_entity (ref 193))

86 (86 type explicit_attr
attribute_decls (“where_clause_entity”)
is_optional true
base_type_entity (ref 163))

87 (87 type explicit_attr
attribute_decls (“wrapped_type_decl”)
is_optional nil
base_type_entity (ref 194))

88 (88 type explicit_attr
attribute_decls (“labelled_expressions”)
is_optional nil
base_type_entity (ref 195))

89 (89 type supertype_declaration
is_abstract true
supertype_expression_entity (ref 90))

90 (90 type supertype_expression
supertype_factor_entity (ref 91)
supertype_continuations nil)

91 (91 type one_of
supertype_expressions ((ref 95)
(ref 97) (ref 98)))

92 (92 type supertype_entity_ref
referenced_type (ref 12))

93 (93 type supertype_entity_ref
referenced_type (ref 25))

94 (94 type supertype_entity_ref
referenced_type (ref 32))

95 (95 type supertype_expression
supertype_factor_entity (ref 92)
supertype_continuations nil)

96 (96 type supertype_entity_ref
referenced_type (ref 18))

97 (97 type supertype_expression
supertype_factor_entity (ref 93)
supertype_continuations nil)

98 (98 type supertype_expression
supertype_factor_entity (ref 94)
supertype_continuations nil)

99 (99 type supertype_entity_ref
referenced_type (ref 29))

100 (100 type supertype_entity_ref
referenced_type (ref 40))

101 (101 type supertype_entity_ref
referenced_type (ref 27))

102 (102 type supertype_entity_ref
referenced_type (ref 22))

103 (103 type supertype_expression
supertype_factor_entity (ref 96)
supertype_continuations nil)

104 (104 type supertype_expression
supertype_factor_entity (ref 99)
supertype_continuations nil)

105 (105 type supertype_expression
supertype_factor_entity (ref 100)
supertype_continuations nil)

106 (106 type supertype_expression
supertype_factor_entity (ref 101)
supertype_continuations nil)

107 (107 type supertype_expression
supertype_factor_entity (ref 102)
supertype_continuations nil)

108 (108 type one_of
supertype_expressions ((ref 103)
(ref 104) (ref 105)
(ref 106) (ref 107)))

109 (109 type supertype_expression
supertype_factor_entity (ref 108)
supertype_continuations nil)

110 (110 type supertype_declaration
is_abstract true
supertype_expression_entity (ref 109))

111 (111 type supertype_entity_ref
referenced_type (ref 19))

112 (112 type supertype_entity_ref
referenced_type (ref 41))

113 (113 type supertype_expression
supertype_factor_entity (ref 111)
supertype_continuations nil)

114 (114 type supertype_expression
supertype_factor_entity (ref 112)
supertype_continuations nil)

115 (115 type one_of
supertype_expressions ((ref 113) (ref 114)))

116 (116 type supertype_expression
supertype_factor_entity (ref 115)
supertype_continuations nil)

117 (117 type supertype_declaration
is_abstract true
supertype_expression_entity (ref 116))

118 (118 type supertype_entity_ref
referenced_type (ref 33))

119 (119 type supertype_expression
supertype_factor_entity (ref 118)
supertype_continuations nil)

120 (120 type supertype_entity_ref
referenced_type (ref 34))

121 (121 type supertype_expression
supertype_factor_entity (ref 120)
supertype_continuations nil)

122 (122 type one_of
supertype_expressions ((ref 119) (ref 121)))

123 (123 type supertype_expression
supertype_factor_entity (ref 122)
supertype_continuations nil)

124 (124 type supertype_declaration
is_abstract true
supertype_expression_entity (ref 123))

125 (125 type supertype_entity_ref
referenced_type (ref 37))

126 (126 type supertype_entity_ref
referenced_type (ref 26))

127 (127 type supertype_entity_ref
referenced_type (ref 38))

128 (128 type supertype_expression
supertype_factor_entity (ref 125)
supertype_continuations nil)

129 (129 type supertype_expression
supertype_factor_entity (ref 126)
supertype_continuations nil)

130 (130 type supertype_expression
supertype_factor_entity (ref 127)
supertype_continuations nil)

131 (131 type one_of
supertype_expressions ((ref 128)
(ref 129) (ref 130)))

132 (132 type supertype_expression
supertype_factor_entity (ref 131)
supertype_continuations nil)

133 (133 type supertype_declaration
is_abstract true
supertype_expression_entity (ref 132))

134 (134 type supertype_entity_ref
referenced_type (ref 20))

135 (135 type supertype_entity_ref
referenced_type (ref 31))

136 (136 type supertype_expression
supertype_factor_entity (ref 134)
supertype_continuations nil)

137 (137 type supertype_expression
supertype_factor_entity (ref 135)
supertype_continuations nil)

138 (138 type one_of
supertype_expressions ((ref 136) (ref 137)))

139 (139 type supertype_expression
supertype_factor_entity (ref 138)
supertype_continuations nil)

140 (140 type supertype_declaration
is_abstract true
supertype_expression_entity (ref 139))

141 (141 type supertype_entity_ref
referenced_type (ref 13))

142 (142 type supertype_entity_ref
referenced_type (ref 42))

143 (143 type supertype_expression
supertype_factor_entity (ref 141)
supertype_continuations nil)

144 (144 type supertype_expression
supertype_factor_entity (ref 142)
supertype_continuations nil)

145 (145 type one_of
supertype_expressions ((ref 143)
(ref 144)))

146 (146 type supertype_expression
supertype_factor_entity (ref 145)
supertype_continuations nil)

147 (147 type supertype_declaration
is_abstract true
supertype_expression_entity (ref 146))

148 (148 type type_decl_wrapper
wrapped_type_decl (ref 2))

149 (149 type entity_block_wrapper
wrapped_entity_block (ref 15))

150 (150 type simple_types
type_name string_boolean)

151 (151 type entity_block_wrapper
wrapped_entity_block (ref 13))

152 (152 type type_decl_wrapper
wrapped_type_decl (ref 8))

153 (153 type type_decl_wrapper
wrapped_type_decl (ref 9))

154 (154 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 155)
is_optional nil
is_unique nil
base_type_entity (ref 156))

155 (155 type bound_spec
bound_1 “1”
bound_2 “?”)

156 (156 type entity_block_wrapper
wrapped_entity_block (ref 18))

157 (157 type entity_block_wrapper
wrapped_entity_block (ref 36))

158 (158 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 159)
is_optional nil
is_unique nil
base_type_entity (ref 160))

159 (159 type bound_spec
bound_1 “0”
bound_2 “?”)

160 (160 type entity_block_wrapper
wrapped_entity_block (ref 21))

161 (161 type entity_block_wrapper
wrapped_entity_block (ref 17))

162 (162 type entity_block_wrapper
wrapped_entity_block (ref 23))

163 (163 type entity_block_wrapper
wrapped_entity_block (ref 46))

164 (164 type entity_block_wrapper
wrapped_entity_block (ref 44))

165 (165 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 166)
is_optional nil
is_unique nil
base_type_entity (ref 153))

166 (166 type bound_spec
bound_1 “1”
bound_2 “?”)

167 (167 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 168)
is_optional nil
is_unique nil
base_type_entity (ref 169))

168 (168 type bound_spec
bound_1 “1”
bound_2 “?”)

169 (169 type type_decl_wrapper
wrapped_type_decl (ref 4))

170 (170 type type_decl_wrapper
wrapped_type_decl (ref 7))

171 (171 type type_decl_wrapper
wrapped_type_decl (ref 6))

172 (172 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 173)
is_optional nil
is_unique nil
base_type_entity (ref 174))

173 (173 type bound_spec
bound_1 “1”
bound_2 “?”)

174 (174 type entity_block_wrapper
wrapped_entity_block (ref 38))

175 (175 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 176)
is_optional nil
is_unique nil
base_type_entity (ref 177))

176 (176 type bound_spec
bound_1 “0”
bound_2 “?”)

177 (177 type entity_block_wrapper
wrapped_entity_block (ref 28))

178 (178 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 179)
is_optional nil
is_unique nil
base_type_entity (ref 180))

179 (179 type bound_spec
bound_1 “0”
bound_2 “?”)

180 (180 type entity_block_wrapper
wrapped_entity_block (ref 45))

181 (181 type entity_block_wrapper
wrapped_entity_block (ref 16))

182 (182 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 183)
is_optional nil
is_unique nil
base_type_entity (ref 184))

183 (183 type bound_spec
bound_1 “0”
bound_2 “?”)

184 (184 type entity_block_wrapper
wrapped_entity_block (ref 14))

185 (185 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 186)
is_optional nil
is_unique nil
base_type_entity (ref 187))

186 (186 type bound_spec
bound_1 “1”
bound_2 “?”)

187 (187 type entity_block_wrapper
wrapped_entity_block (ref 25))

188 (188 type type_decl_wrapper
wrapped_type_decl (ref 10))

189 (189 type entity_block_wrapper
wrapped_entity_block (ref 39))

190 (190 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 191)
is_optional nil
is_unique nil
base_type_entity (ref 192))

191 (191 type bound_spec
bound_1 “0”
bound_2 “?”)

192 (192 type entity_block_wrapper
wrapped_entity_block (ref 35))

193 (193 type entity_block_wrapper
wrapped_entity_block (ref 43))

194 (194 type entity_block_wrapper
wrapped_entity_block (ref 40))

195 (195 type aggregation_types
aggregation_type_name string_list
bound_spec_entity (ref 196)
is_optional nil
is_unique nil
base_type_entity (ref 197))

196 (196 type bound_spec
bound_1 “1”
bound_2 “?”)

197 (197 type entity_block_wrapper
wrapped_entity_block (ref 24)))))