Considerations for the Transformation of STEP Physical Files

Robert Kohout
edited by Stephen Clark

July 12, 1993

A Note from the Editor: The work described was performed by Robert Kohout in the summer of 1990, at which time this paper was drafted. Due to renewed interest in the problem, it seems appropriate to release this heretofore unpublished work so that others may refer to it. To this end, I have performed my editorial duties, in truth passing the paper on much as I found it. The entire technical content is due to Mr. Kohout; editorial errors are, of course, solely mine.

1 Introduction

The emerging Standard for the Exchange of Product Model Data, [1] commonly referred to as STEP, is an international standardization effort which addresses the need to share data in a complex computer environment. STEP will consist of three primary components of interest to the present work: an information modeling language called EXPRESS [SPIB]; a set of information models specified in EXPRESS which define a class of information to be exchanged among STEP-based applications; and an exchange file syntax which allows data conforming to an EXPRESS information model to be exchanged between applications [VANM]. In the future, there will be other implementation forms as well, such as a functional interface to a persistent database form [FOWL]. The various information models, however, remain independent of these implementation forms.

PDES (Product Data Exchange using STEP) is the name given to the United States' activity in support of the development of STEP. As a part of this activ-

[1] The Standard for the Exchange of Product Model Data (STEP) is a project of the International Organization for Standardization (ISO) Technical Committee on Industrial Automation Systems (TC184) Subcommittee on Industrial Data and Global Manufacturing Programming Languages (SC4). For an overview of the standard refer to *Part 1: Overview and Fundamental Principles* [MASO].

ity, the National PDES Testbed has been established at the National Institute of Standards and Technology [2]. The Testbed's mission is to accelerate the development of STEP and to act as a resource for US industry in the validation and conformance testing of STEP. The Testbed develops and maintains software for the validation and testing of STEP, as well as serving as a repository of PDES and STEP documents.

Because the software environment at the Testbed deals with the testing and validation of an emerging standard, it must be able to accomodate changes to various parts of the specification. To this end, the software architecture is quite modular [CLAR, MORR]. An embeddable EXPRESS parser, Fed-X [LIBE], isolates programs from the syntactic details of EXPRESS. In addition, it allows most of the rest of the Testbed software to be written in a generic, schema-independent fashion. Thus, for example, database loaders first read an EXPRESS schema before attempting to interpret any STEP physical files.

An additional problem in this environment of a constantly changing specification is that of keeping test data up-to-date with respect to changing schemas. Whenever a change is made to a conceptual model, the possible need to change existing data instantiations based on the old model arises. This document discusses some of the salient issues involved in automating the translation of such data. In general, a change in the underlying model will motivate a change in any existing form of data representation. In the Testbed environment, this currently implies changes in relational databases, STEP physical files, and memory-resident working form representations. For the purposes of this discussion, we will concentrate on the physical file, as this is the only exchange form to be included in STEP Version 1.0. Each method of data instantiation has its own nuances; however, for the most part the issues we discuss apply more or less equally well to each.

2 Motivations for Change

We can identify three major classes of changes to parts of STEP which may have an impact on the validity of existing physical files. In descending order of their (apparent) relative frequency, these are

- **Changes to the Conceptual Model.** These changes are typically made by a standards committee and are due only to design considerations with respect to the model itself.

- **Changes to the EXPRESS Language.** Since the STEP models are defined using EXPRESS, a change in either the syntax or semantics of the language may necessitate a change in the model itself.

[2]Funding for the Testbed has been provided by the Computer-Aided Acquisition and Logistic Support (CALS) program of the Office of the Secretary of Defense. The work described is funded by the United States Government and is not subject to copyright.

- **Changes to the Physical File Specification.** If the formal specification of the STEP physical file format changes, existing files must be modified to reflect this change.

These distinctions are somewhat artificial. For example, changes in the model due to changes in EXPRESS may be thought of as a special case of the first category. A naive approach to detecting changes in the conceptual model that simply compared one version of the model to another and attempted to determine the differences does not neccesarily have to consider the reasons for a change. We make this distinction primarily because it is our opinion that any reasonable physical file translator will have to be able to parse the EXPRESS language, as well as the STEP physical file. For this reason, the distinction is important. Changes in a STEP file due only to changes in the model should be easier to process than those due to changes in the EXPRESS language, since the latter will require modifications to the EXPRESS parser.

3 Codifying Changes in the Model

To translate existing files effectively, we need some representation for the transformation that is to be applied to them. Some interest has been expressed in generating such a representation automatically, so we will address the possibility of doing so.

3.1 An Example from Computer Language Translation

To make use of a familiar analogy, consider a typical compiler for a computer language. Such a program makes a translation from one language (relatively) easily understood by humans to another easily interpreted by the computer (i.e. machine language). The translation is generally deterministic and unambiguous, and is made in a well understood, well documented fashion. We can subdivide such a translator into two parts: the parser, which recognizes constructs in the higher-level source language, and the code generator which produces corresponding machine language code. In practice, parsing is a very well understood, almost automatic task, while code generation is somewhat more of an art form. There exist a plethora of "parser generators" to take a specification of the syntax for a high level language and produce code that will effectively parse programs written in that language. No such program exists to produce code generators that is sufficiently powerful and general to be useful in real-life applications.

The reasons for this fact are unclear and subject to much debate; however, it seems safe to say that while it is relatively easy to recognize the constructs of a source language, it is not nearly so simple to specify the *transformation* we must apply to that construct to produce the equivalent construct in the target language. This task usually requires a significant use of human intelligence.

```
      block structured                              non-block structured

procedure :==                           |     procedure :==
  PROCEDURE id ( var-list)              |      PROCEDURE id (var-list)
                                        |
     local-variables ;                  |         local-variables ;
                                        |
   [ procedure ]                        |
                                        |
    BEGIN                               |       BEGIN
                                        |
    statement-list ;                    |       statement-list ;
                                        |
    END                                 |       END
```

Figure 1: Syntax For Block Structured and Non-Block Structured Procedure

Generally speaking, the problem derives from the fact that relatively small syntactic changes can imply arbitrarily large semantic differences. A 'classical' example can be seen in the attempt to translate a block structured language into a non-block structured language (e.g. Pascal to C). The syntactic difference is illustrated in Figure 1.

While our block structured example is not as robust as Pascal, which allows for more than one procedure to be declared within a block, this example is sufficient for our purposes. The syntactic difference between these two constructs is slight : we have only allowed for the possible nesting of a single procedure definition in the block structured case. On the other hand, the semantic implications are considerable. In particular, to translate from the block structured language into the non-block structured language, our transformation has to implement the lexical scoping rules of the block structured language explicitly. What was once a feature of the compiler has to be mapped out as a feature of the output program. There is no way to quantify the difficulty this presents, but from the author's own anecdotal experience this feature of Pascal is the single most difficult problem in translating from Pascal to C. The important point is that it is quite difficult for a human to devise a means of making this transformation, and yet it is based on a very small syntactic difference.

3.2 An Example from STEP

To come a little bit closer to home, let us examine the problem of representing enumeration values in STEP exchange files. [3] In an early version of the exchange file mapping [ALTE], enumeration values were implemented as 0 based integers. Thus the enumeration (RED, GREEN, BLUE) mapped to (0, 1, 2), and an attribute of this type in a particular instance in a physical file would have one of these three integer values. Suppose that the model were changed to add the color BLACK to this enumeration type. If we simply made the new type (RED, GREEN, BLUE, BLACK) we would have to make no changes whatsoever in existing physical files, since they presumably only have entries values 0, 1, or 2 and these values still have the same semantic interpretation. However, if the modeler decided to make the list (RED, BLACK, GREEN, BLUE), then suddenly the transformation becomes more complex: now all attributes with value 0 remain 0, those with 1 become 2, and those with 2 become 3. Any program that is going to be able to handle the general problem of specifying transformations between versions of a model will have to be able to handle this case.

That may not seem to be much of a problem, but consider the following: suppose we assume a later version of the specification (e.g., the current version, [VANM]) when we design our translator. This version implements the same enumeration type above as .RED., .GREEN. or .BLUE. in a particular instantiation of the type, so we wouldn't have the problem we described above. However, if at some later point in time the standard reverted to the earlier version of the physical file format (and we can't assume that it won't and remain completely general) then our translator would have to somehow be capable of suddenly producing the transformation described in the preceding paragraph without having had any *a priori* knowledge of such a situation. [4]

3.3 Things Could be Even Worse

Upon consideration of the problem presented in the previous subsection, one may well suggest that we can solve it by forbidding modelers from adding enumeration types in the middle of an enumeration list. This would solve the present problem, but previously documented changes [YANG] suggest that the above example is an overly *simplistic* case of a more general problem. When the data modeler changes a STEP model, a complex transformation between the old and new models may exist. Providing a *general purpose* solution that could produce these transformations automatically appears quite difficult; indeed, it seems that the problem would first require the definition of a complete

[3] While the current implementation of the physical file may not require a solution to this problem, a fully general translator should be able to handle this case.

[4] This particular problem may be surmountable by requiring changes in the physical file structure to provide an algorithm that could produce the appropriate transformation when required.

formal semantic specification for both EXPRESS and the particular information model. This is certainly a desirable goal, but the effort involved is not justified by the purposes of the current project alone. More time would be spent in the definition of the formal semantics than could possibly be saved by being able to translate the data automatically.

In summary, the hope for a program that could take any two versions of an EXPRESS model and generate the transformations necessary to translate the physical files of one model into those of the other is unrealistic at this time. This sort of solution, which we refer to as a *completely general* solution, may be desirable, but it is not practical. A more reasonable design will be less robust, in the sense that it will not be able to produce all the requisite transformations automatically, but it may nonetheless be sufficient for our purposes.

4 What Can be Done

Assuming that we have abandoned hope for a fully general solution to the problem of translating physical files, we can then consider 'partial' solutions. Such solutions tend to be heuristic in nature; their proper design and implementation is heavily dependent upon the nature of the particular problem they are intended to solve. Thus an empirical analysis of the types of translations to be made and their relative frequencies is in order. To begin, we list the types of motivating changes presented in section 2 and add some assertions regarding the ease with which the resulting transformations can be handled.

1. Transformations resulting from the change in the format in physical files are relatively simple in general. Indeed, we will argue later that these changes should properly be handled separately from transformations motivated by changes in the models, and that this approach both simplifies the resulting tools and makes them more powerful. Most of the changes in this class are similar to the change in the representation of enumeration values discussed above.

2. Transformations resulting from changes in the EXPRESS language itself also tend to be relatively straightforward, although these changes may result in significant changes to the information models. The latter might well have considerable impact on the contents of existing physical files.

3. Transformations resulting from changes in the conceptual model may be arbitrarily complex.

In the remainder of this section, we will focus on the final category, as it is the most useful and the most interesting. A brief proposal for a solution to the first category is presented in section 4.2. For the purposes of the current study, we will assume that transformations motivated by changes to the EXPRESS language will be dealt with as a special case of transformations resulting from changes to the underlying information models.

A more careful analysis reveals that we can break the third group into categories, in descending order of frequency, as follows:

- **Adding or Deleting an Attribute.** This is by far the most common change made to conceptual models, and is a very simple transformation to either detect or specify. A slightly more difficult case occurs when the change is actually made to a supertype, in which case it must be propagated to all subtypes instantiated in the physical file.
- **Attribute Shifts.** Occasionally an attribute will be shifted to a new position in an entity definition, or from one entity to another. One can imagine a rather straightforward, but imperfect scheme, for detecting this sort of change. Again, this should be a rather easy change to specify.
- **Change of Type.** While this can happen generally, the most common and problematic changes involve aggregates. For example, the modeler may want to change the cardinality of a set, or perhaps change a list to a set. These changes can easily be detected, but the transformations are not always easy to specify. For example, when changing an attribute from a set to a list, we generally don't have enough information to produce a properly ordered list.
- **Complex Transformations.** We include in this class those transformations not described above, with the understanding that there may exist some 'simple' transformations that we have yet to enumerate. An earlier study [YANG] provides a good example of a member of this class. In an effort to remove redundant data, the ISO Integration Committee decided to radically alter the structure and relationship of several entities. As we have tried to show above, such transformations cannot generally be detected, nor are they easily specified.

Given this list, we see that the main problems come from a relatively small but important class of changes. Except for those changes in the last category, it may be possible to process EXPRESS model definitions automatically to detect changes and even specify the appropriate transformations. Whether or not to write a translator that tries to do the easy work is a major design decision.

4.1 Proposed Approach

In this section, we outline an approach to the construction of a translator which can apply the transformations listed above to a particular physical file. Whether or not one decides to design a program to automatically detect certain classes of transformations, the proposed translator will still require some means of specifying the transformations to be made. That is, regardless of whether or not the physical file translator obtains its transformations from a program or a human is largely irrelevant. Conceptually, a transformation detector will have to encode the appropriate translation in some intermediate form, which is then interpreted by the translator to alter the physical file. If we make this intermediate form explicit (i.e. external to the program), we then have a language with which to specify transformations. The explicit design of such a language is advantageous for several reasons, some of which are:

- A properly designed language can be used by humans to characterize transformations too complex to determine automatically.
- The existence of an intermediate language would allow for fairly straightforward mergers of transformations detected automatically with those provided by humans. We can design a translator to read in a list of transformations and apply them independently of their source.
- The design and implementation of a transformation detector and the physical file translator can proceed independently. The language which characterizes the intermediate form serves as a complete interface specification.
- A transformation language provides the modeler a means of unambiguously specifying the intended relationship between instances under the old model and those under the new.

On the basis of the above discussion, it seems reasonable that the first step in the design of a physical file transformation system is the design of an intermediate language to represent the required transformations. We have already provided an informal discussion of the requirements: we must be able to specify the addition or deletion of an attribute, the shifting of an attribute from one entity to another, and changes of type. More significantly, we must provide the data modeler a means for specifying the complex transformations which may accompany changes in the conceptual model.

4.2 An Approach for Handling Changes in the Exchange File Format

Assume that we have a specification for data transformations in an intermediate form. The translator will read in this specification, and begin processing some number of physical files, one at a time, by reading them into a temporary working form, applying the necessary transformations, and then writing them back out in physical file format. This last step requires a knowledge of what the appropriate format is. Normally, this would just be the original format, but in the case where the physical file has changed, this could be the new format. The design of the translator will have to anticipate such changes, so it should be designed to take a physical file format specification as an input. This gives us the two-fold advantage of being able to easily modify the translator to reflect changes in the physical file format, and being able to easily map such changes to the files themselves.

This specification of the physical file format, which effectively describes the target language of the translator is problematic. If sufficiently general, we should be able to use this translator to translate from a STEP physical file to *any* output file format. In particular, this would provide a convenient means of translating STEP data into proprietary formats. However, inspection of some of the existing proprietary formats for storing data indicates that a fully general solution to this problem will not be simple. Moreover, there is the additional problem that a STEP conceptual model is typically a superset of a proprietary format, so an appropriate mapping may not always exist. For these reasons, we leave the scope and requirements for this specification an open issue.

4.3 Steps to a Solution

Based upon the above discussions, we can identify the tasks listed below. A reasonable solution will approach these problems in the following order:

1. The design of an intermediate language for specifying transformations between instances of different versions of a conceptual model.

2. The design of a physical file format specification which can be used by the translator to produce properly formatted output.

3. The design and implementation of a translator which can take intermediate form translations, a physical file format specification, and a physical file to produce an updated file in the specified format.

4. The design and implementation of a change detector which can automatically detect many of the simple changes made to a model, such as attribute shifts, additions or deletions and produce an intermediate form representation of the necessary translations.

5 Data Transformation Language

This section presents a preliminary description of the required operations for a language to describe data transformations in STEP exchange working files. It then describes two possible syntactic representations for these operations.

5.1 Required Operations

Any language for specifying data transformations must provide the following operations:

1. **Entity Creation.** New entities may be added to a model at any time, so this facility must be included. Note that the full power of the EXPRESS entity declaration must be supported.

2. **Entity Deletion.** While it may seem that modelers will rarely wish to delete existing entities from a model, preferring instead to rename or redefine existing entities, this operation should nonetheless be available.

3. **Renaming an Entity.** During the integration process in STEP, it is likely that entities will be renamed, whether or not other aspects of their definitions are changed.

4. **Changing the Supertype/Subtype Status of an Entity.** New subtypes are frequently added to supertype entities. Other changes may not be as commonly made to existing entities, but they should nonetheless be supported.

5. **Attribute Creation.** The addition of a new attribute to an existing entity is a very common change. Note that additions to supertypes must naturally propagate to instantiated subtypes.

6. **Attribute Deletion.** A relatively common modification.

7. **Renaming an Attribute.** It is not clear how often such an operation would be used, but the design should support it.

8. **Moving an Attribute Between Entities.** It may be possible to accomplish this task by some combination of other operations, but this is a common enough change that it should be supported directly.

9. **Changing the Type of an Attribute.** This is another frequent change. Some changes are straightforward, others may be more complicated.

10. **Copying Data.** This can be at either the entity or attribute level.

11. **Transforming Data.** The translator should allow arbitrarily complex, functional transformations of existing data to be made. This implies that it must include an interpreter capable of evaluating any EXPRESS expression.

12. **Scope Declarations.** This is to allow for the possibility of disambiguating entities of the same name in different schemas.

13. **Assertions.** This is the only item on this list which may be considered optional. It may be desirable to allow the specification of data constraints to insure data integrity during the transformation.

5.2 Alternative Implementations

In this section, we present two alternative syntactic representations for the operations listed above. In general, the appropriate representation is a matter of taste. Since it should be fairly easy for a program to generate transformations using whatever syntax we describe, a primary design consideration should be ease of use by humans. The first alternative makes the desired operations explicit, while the second is roughly based upon existing EXPRESS constructs. Thus the first may be easier for a neophyte to learn and understand, while the latter may be more familiar to EXPRESS modelers.

In these descriptions, boldface indicates keywords of the proposed language, while italics are intended to stand for uninstantiated instances of a general type. Braces surrounding an object as in [*option*] represent optional constructs, and a vertical bar, |, is used to separate alternatives, which may be grouped in parentheses for clarity. An asterisk, *, indicates that a construct may appear zero or more times.

5.2.1 Alternative A

The constructs for this alternative are:

1. **create entity** *entity* [*type_list*] **as** *attr_list*;

   Where *type_list* allows for sub/supertype declarations and *attr_list* follows the EXPRESS conventions for entity attribute lists.

2. **delete entity** *entity*;

3. **rename entity** *entity* **to** *new_entity*;

4. **entity** *entity* **is (super | sub) type of** *entity*;

5. **insert** *attr* **into** *entity* [((**after** | **before**) *attr2*) | **at** *pos*] **as** *type*;

   Where *type* is a valid EXPRESS type description and *pos* is an integer specifying the absolute position of the attribute in the attribute list of *entity*.

6. **delete attribute** *attr* **from** *entity*;

7. **rename attribute** *attr* **to** *new_attr* **in** *entity*;

8. **move** *attr* **from** *entity* **to** *new_entity* **at** *pos*;

   Where *pos* describes the attribute's position in the new attribute list, as documented above for the insert statement.

9. **type of** *attr* **in** *entity* **is** *type*;

   Where *type* is a valid EXPRESS type declaration.

10. **update** *entity* **set** *attr* **:=** *expr*[, *attr_n* **:=** *expr_n*]* [**where** *cond*] **;**

    Where *cond* is a Boolean function of each entity-attribute instance. Note that this construct can be used for both operations 10 and 11 of section 5.1. That is, it can be used to both copy and transform data.

11. **beginscope** *schema* **;**

    Where *schema* is a valid schema within the current scope. The universal scope is the default, and an active scope must be closed by a matching **endscope** statement.

12. **endscope ;**

    This closes the most recently opened scope.

5.2.2 Alternative B

The constructs in this implementation would be based upon the EXPRESS **map** construct, with some extensions and modifications. Certain EXPRESS constructs, such as entity declarations, could be used as currently implemented. In this section we iterate through the required operations of section 5.1 and show how they would be implemented.

1. **Entity Creation** This is done with an EXPRESS entity declaration.

2. **Entity Deletion.** This is implemented via an extension to the **map** statement of EXPRESS as follows:

    map entity null from *old-entity* **;**
    end_entity ;

3. **Renaming an Entity.** This is also done with a variant of the **map** statement:

    map entity *new-name* **from** *old-name* **;**
    end_entity ;

    but this is ambiguous: should this imply that *old-name* is deleted, or that *new-name* is just a copy of *old-name*? Under the current definition of EXPRESS, the latter is the case. We must therefore either assume that the old entity always is destroyed, or we need some construct to indicate whether the old entity should be retained. For the purposes of this document, we adopt the following convention: **map** will always imply that its second entity argument is to be deleted, and we will use a different construct, **copy**, which will imply that the second entity argument is to be retained unchanged. As much as possible, the syntax and semantics of **map** and **copy** will otherwise be identical.

4. **Changing the Supertype/Subtype Status of an Entity.** In general, when we map one entity to another, we want to keep the declaration of the base entity. However, when a model changes, we may have to alter these declarations. We can use **map** as follows:

   map entity *new* **from** *old* **supertype of (**a, b**);**
   end_entity;

   map entity *new* **from** *old* **subtype of (**a**);**
   end_entity;

   map entity *new* **from** *old* **subtype of (**a**) supertype of (**b**);**
   end_entity;

   The general syntax for the sub/supertype field is the same as that of EXPRESS. Note that is the case where the destination is the same, **map** and **copy** would have the same functionality.

5. **Attribute Creation.** Here we require an extension to the EXPRESS version of **map**, as well as a redefinition of some semantics. [5] In EXPRESS, a mapping is made of all named attributes, and those which are not named explicitly are assumed to be deleted. We will make the opposite assumption, i.e., that all the attributes of the second entity argument will be mapped unchanged to the first unless it is explicitly stated otherwise. Thus

   map entity *new* **from** *old* **;**
   end_entity;

   works as expected and is only a special case of a more general principle. We can then add an attribute by simply naming it and giving its type, as in:

   map entity *new* **from** *old* **;**
   new_attribute **: integer(7) ;**
   end_entity ;

   Note that this syntax does not allow us to specify the position at which the attribute is to be added. This may not be sufficient.

[5] As we move further away from EXPRESS, it becomes clearer that perhaps **map** should not be used, but rather some other name or construct. This document is not intended as a final design. It was suggested that we examine using **map**; here we try to take its use as far as we can.

6. **Attribute Deletion.** Given the assumption above, we can then delete an attribute with the following constuct:

   map entity *new* **from** *old* ;
   attribute **is deleted ;**
   end_entity;

7. **Renaming an Attribute.** Again, we need to amend EXPRESS.

   map entity *new* **from** *old* ;
   attribute **is renamed to** *new_name* ;
   end_entity;

8. **Moving an Attribute Between Entities.** Since this command affects two entities at once, it doesn't mesh well with the **map** construct, which generally affects only one entity (and deletes another). Thus we suggest the syntax from the above alternative. That is:

   move *attr* [, *attr*]* **from** *entity_1* **to** *entity_2* [**at** *pos*] ;

9. **Changing the Type of an Attribute.** Here we use the existing EXPRESS syntax:

   map entity *new* **from** *old* ;
   attribute **: retype as** *type* ;
   end_entity ;

10. **Copying Data.** We need to extend the **map** statement so that it can refer to more than one entity. To do this, we allow attributes to be referenced as *entity.attribute* in expressions and where clauses. For example,

    map entity *entity* **from** *entity* ;
    attribute **:=** *entity_x.attribute_y* ;
    where *cond* ;
    end_entity ;

    Note that *cond* must be powerful enough to indicate which instances of *entity_x* correspond to which instances of *entity*. This is a difficult problem, and remains an open issue.

11. **Transforming Data.** We can use the same basic construct that we use above, and extend it to allow arbitrary expressions on the right-hand side of assignments. If we allow an optional type declaration to follow an expression, we can increase the expressive power of the language. That is, we can allow statements such as:

map entity *new* **from** *old* ;
attribute_a := *entity_x.attribute_y* / **PI** : **real(9)** ;
attribute_b := *old.attribute_b* + *old.attribute_c* ;
attribute_c **is deleted** ;
end_entity ;

12. **Scope Declarations.** Here we suggest the use of the **beginscope** and **endscope** declarations introduced in Alternative A.

5.3 Comparing Alternatives

Briefly, the primary advantage to the first alternative language is its simple syntax. A person using this language to express transformations will find it relatively easy to learn, understand and use. It may also be easier to generate statements in this language automatically. At the very least, automatically generated statements in this language should be more idiomatic. [6]

The second alternative offers three advantages. First, it has more expressive power. One statement in this language can accomplish much more than a single statement of Alternative A. Secondly, since the language is based primarily upon existing EXPRESS constructs, one can hope that it would be easily mastered by a modeler seeking to describe conceptual model changes. Thirdly, from an implementation standpoint, the lexical analyzer for this language is effectively written already, as is a portion of the parser, since both can be borrowed from existing EXPRESS compilers. There is less new language to parse, so the parser should be simpler.

6 Summary

In the validation and testing environment of the National PDES Testbed, the problem frequently arises of migrating test data from one version of a schema to another, or from one version of the STEP exchange file format specification to another. We have discussed various issues arising from this problem, and have proposed an approach to a solution. The basic approach is to define a concrete intermediate representation for the changes between two schemas; this representation can then be written by a modeler making changes to a model, or might perhaps be automatically generated by some difference detection engine. Changes in the exchange file format can be dealt with by appropriate manipulation of the output phase of the transformation engine.

> *Editor's note:* After writing this paper, Mr. Kohout built an experimental system based on the EXPRESS-based (Alternative B)

[6] This is just another way of saying that an automatic transformation generator would tend to find relatively simple changes, and generate simple statements. The greater expressive power afforded by Alternative B would most likely not be taken advantage of.

syntax. Although his time at NIST ran out before the system was put into production, he was able to demonstrate the viability of his approach. Future work at the Testbed on this problem will likely be based on this approach.

7 References

[ALTE] Altemueller, J., ed., *The STEP File Structure*, Working Draft N241, ISO TC184/SC4/WG1, February 1988.

[CLAR] Clark, S. N., *An Introduction to The NIST PDES Toolkit*, NISTIR 4336, National Institute of Standards and Technology, Gaithersburg, MD, May 1990.

[FOWL] Fowler, J., ed., *Proposal for the STEP Data Access Interface Specification*, Working Draft, ISO TC184/SC4/WG7, January 28, 1992.

[LIBE] Libes, D., and S. Clark, *Fed-X: The NIST EXPRESS Parser*, NISTIR, National Institute of Standards and Technology, Gaithersburg, MD, forthcoming.

[MASO] Mason, H., ed., *ISO 10303 Industrial Automation Systems – Product Data Representation and Exchange – Part 1: Overview and Fundamental Principles*, Working Draft N43, ISO TC184/SC4/WGPMAG, October 7, 1991.

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

[SPIB] Spiby, P., ed., *ISO 10303 Industrial Automation Systems – Product Data Representation and Exchange - Part 11: The EXPRESS Language Reference Manual*, Committee Draft N14, ISO TC184/SC4, April 29, 1991.

[VANM] Van Maanen, J., ed., *ISO 10303 Industrial Automation Systems – Product Data Representation and Exchange - Part 21: Clear Text Encoding of the Exchange Structure*, Committee Draft, ISO TC184/SC4, March 12, 1991.

[YANG] Yang, Y., *IPO Integration Committee and SG7 Joint Meeting Minutes 6 June 90*, June 21, 1990.