Dynamic Objects and Meta-level Programming of an EXPRESS Language Environment

Peter Denno

Manufacturing Systems Integration Division
National Institute of Standards and Technology
Gaithersburg, Maryland, USA

pdenno@cme.nist.gov

(301) 975-3595

ABSTRACT

This paper describes design and programming techniques employed in the development of a language environment for the EXPRESS information modeling language. A fundamental concern in the development of language environments for object flavored languages is the degree to which the object model of the implementation language matches that of the language being modeled. If there is significant mismatch, the programmer is forced to reconcile the differences with little aid from the implementation language's object model, *i.e.* eschewing the implementation's native object model and programming with elementary tools the object oriented behaviors of the target language environment. This paper describes how object model mismatch was eliminated and a responsive, incremental EXPRESS language environment is being developed using the Common Lisp Object System (CLOS) metaobject protocol (MOP) and dynamic object techniques.

1.0 INTRODUCTION

The paper begins with a presentation of background information regarding EXPRESS and its role in ISO 10303, more commonly referred to as STEP (The Standard for the Exchange of Product Model Data). Section 2.1 provides an overview of design principles commonly employed in the design of EXPRESS language tools. Sections 2.2 and 2.3 concern the problems of object model mismatch that confront designers of EXPRESS tools. Section 3.0 presents aspects of the author's Common Lisp implementation of an EXPRESS language environment. It describes solutions to the problems introduced in section 2 employing dynamic object techniques, CLOS and its metaobject protocol. Section 4.0 describes additional aspects of the EXPRESS language environment currently under development. The paper concludes with a summary of how dynamic object techniques and meta-level programming have contributed to the development of an EXPRESS language environment and how these same techniques might find use in the larger context of STEP development.

2.0 BACKGROUND

EXPRESS [ISO-11], [Schenck] is a formal language used to describe information models of STEP. The STEP standards support the unambiguous communication of industrial information exchanged in forms derived from EXPRESS language information models.[1] Within the STEP architecture, a *product model*, that is, an information model from which a range of similar artifacts can be described, is defined by an *application protocol* (for example, the application protocol, *Ship Structures*, [ISO-218]). Application protocols rely on libraries of commonly used concepts called *integrated resources*. The integrated resources collectively describe an abstract model of products. Components of the abstract model include, for example, *Geometric and Topological Representations* [ISO-42], which can be used in the description of application protocol for ship structures, automotive components, *etc.*. The reuse of common components improves the quality and development efficiency of application protocols and facilitates information sharing among related disciplines (*e.g.,* design systems and production scheduling systems may share an entity authorizing the release of a part for production).

Integrated resources and application protocols are defined in the EXPRESS language. Further discussion of the STEP architecture [ISO-1] is outside the scope of this paper.

[1] EXPRESS itself is an ISO standard developed by the STEP community.

In order to convey the information intended, systems that exchange data must agree upon the semantics of that data. A fundamental design goal of an information modeling language, such as EXPRESS, is to provide the means to describe information models (and subsequently data) reflecting a mutually understood semantics. That is, the language is used to define constraints which data sets exchanged must satisfy. In EXPRESS these constraints take several forms:

- **Constraints on an attribute's type[2] [3]:** the value of an attribute must be type-compatible with the type declared in the attributes's declaration.
- **Constraints on semantic consistency of entity instances:** Rules called *where rules* can be associated with an entity type. For example, an entity type named **`unit-vector`** possessing **`REAL`** valued attributes **`x`** and **`y`** might have a where rule requiring **`x**2 + y**2 = 1`**. These rules are procedurally defined in EXPRESS and may rely on built-in and user-defined functions and procedures.
- **Constraints on an entity instance's type:** EXPRESS provides a flexible mechanism for defining and composing hierarchical entity types (called *complex entity types*) and constraints on the legal composition of types within the hierarchy. For example, an abstract type **`person`** might have subtypes **`male`** and **`female`**. An instance of **`person`** must be one of **`male`** or **`female`** but not both.
- **Constraints on populations of entity instances:** Populations of entities (data sets) must satisfy constraints described in *global rules*. Such rules can, for example, constrain the cardinality of the instances of a type (*e.g.,* there is only one CEO) or require specific relationships between entities (*e.g.*, every employee has a supervisor).
- **Constraints on the existence of an entity instance:** The existence on an entity may depend on the existence of another entity.

2.1 Classification of EXPRESS Language Environments

A language environment for EXPRESS, that is, an environment that allows the development of EXPRESS information models and sample data sets, can be of great value to developers of EXPRESS information models. The EXPRESS for some STEP standards can run into the thousands of lines of code. Since the development of STEP standards has proved to be a lengthy process and a large part of that effort involves the articulation of an EXPRESS information model, the STEP community is quite interested in tools that improve the efficiency of EXPRESS development.

Several tools supporting EXPRESS development exist, [Wilson], [Libes], [Morris], [STI], [Kiekenbeck]. The focus of these tools is more the validation of data sets, the exploration of existing models, and the generation of application interfaces than it is the incremental development of new EXPRESS models. The work described here has as its goal the development of a responsive, incremental development environment for the concurrent development of EXPRESS models and representative data sets, (such an environment as is available to Lisp programmers).

The design of language environments include strictly interpreted approaches, strictly compiled approaches, and designs that utilize both interpretation and compilation. To date, compiled approaches to EXPRESS tools ([Morris], [STI], [Kiekenbeck]) use the subject EXPRESS to automatically generate code in an implementation language (*e.g.,* C++) which is then compiled and link-edited with auxiliary libraries that are independent from the subject code. Compiled approaches to EXPRESS environments have advantages over interpreted approaches in that libraries of compiled EXPRESS can be saved for later inclusion in information models without incurring the cost of interpretation. Also, the automatically generated source code is often useful in various applications that want access to data in the form defined by the subject EXPRESS model. Compiled approaches where compilation and link-edit are lengthy, however, can not support a responsive, incremental development environment: complete processing of large EXPRESS models using this approach can take 30 minutes or more.

A strictly interpreted approach can be more responsive to the incremental development of the EXPRESS subject code but can not take advantage of compiled libraries of supporting EXPRESS information models. Nor can interpreters implemented in statically compiled languages use the constructs of the language (*e.g.,* classes, iteration, arithmetic operations) in direct ways. For example, an interpreter can not generate and use a C++ class to represent an EXPRESS entity type. For these reasons, the development of an interpreter requires the development of fundamental supporting mechanisms unnecessary in compiled approaches. Therefore, the development of an interpreter for a language as large as EXPRESS is a daunting task. Recent developments in the design of interpreters, however, might make interpreted approaches more feasible, (*e.g.,* [Liang]).

Approaches that use both compilation and interpretation are commonly implemented in languages possessing resident programming environments, such as Smalltalk or Lisp.

2.2 Mismatch of Object Models

EXPRESS is an information modeling language, not a programming language. EXPRESS is 'object flavored' but not strictly object oriented. (It does not encapsulate state with methods). EXPRESS makes a distinction between classes and instances, allows class hierarchies and inheritance of data attributes.

An *object model* is a model with a class/instance distinction, encapsulation of state via methods and inheritance of behavior from

[2] *Attributes* correspond to C++ data members or CLOS slots. Attributes are associated with *entities*, analogous to C++ or CLOS classes.

[3] EXPRESS defines the usual primitive types and enumeration and allows user-defined types that generalize or specialize on primitive or other user-defined types

parent classes. *Concrete object models,* those underlying programming languages, vary with respect to inheritance, encapsulation, information hiding, dynamic capabilities and various other features [Manola]. A fundamental concern in the development of a language environment for an object flavored language is the degree to which the object model of the implementation language matches that of the language being modeled. If there is significant mismatch between the object model of the implementation language and that of the target language, the programmer is forced to reconcile the differences with little aid from the implementation language's object model. That is, software is not reused, but rather, beginning again with elementary tools, the programmer implements the object oriented behaviors of the target language environment. This, of course, is a great amount of work for what might be a small (but crucial!) deviation towards the behaviors of the target object model. This paper hopes to illustrate the value of a metaobject protocol (specifically the CLOS metaobject protocol) in reconciling the differences between the object models of the target and implementation languages.

2.3 Three Challenges in Object Model Mismatch[4]

The EXPRESS object model[5] differs in some important ways from the concrete object models underlying common programming languages (C++, Smalltalk, CLOS). Significant areas of mismatch between these languages and EXPRESS are found in the inheritance of attributes (C++ data members, or CLOS slots) and the approach to subclassing. Three characteristics of the EXPRESS language that present challenges in the development of an EXPRESS language environment are described below:

- ***CHALLENGE(same-named-attr):*** When same-named attributes of two EXPRESS entities are inherited through two distinct paths (EXPRESS allows multiple inheritance) two distinct attributes are inherited by the subclass, one from each parent defining the attribute.
- ***CHALLENGE(redefined-attr):*** In EXPRESS an attribute defined in a supertype may be redefined in the subtype only if the domain of values of the subtype is a restriction of the domain of the supertype. Thus an attribute of type **`number`** in the supertype can be redefined as type **`integer`** in the subtype, but an **`integer`** in the supertype can not be redefined as a **`number`** in the subtype.
- ***CHALLENGE(ANDOR-subtyping):*** Unless explicitly prohibited by a clause in the entity definition, the direct subtypes of an EXPRESS entity type can be combined to define additional subtypes.

[4]These are, of course, challenges for compiled approaches. Interpreted approaches using static implementation languages can not make use of the objects in these ways.

[5]Although not strictly object oriented, EXPRESS possesses enough of the characteristics of an object oriented language to make use of the term *object model* meaningful in this context.

An example of this behavior (called *ANDOR subtyping* here) is illustrated in the following example EXPRESS.

```
ENTITY a SUPERCLASS OF (b ANDOR c ANDOR d);
```

This clause declares that the entity combinations a, a+b, a+c, a+d, a+b+c, a+b+d, a+c+d and a+b+c+d are all instantiable, where a+b+c, for example, denotes an anonymous class (*complex entity type*) possessing the attributes (data members, slots) of the classes a, b and c.

Very different solutions to these challenges may be found depending on the implementation language chosen. *CHALLENGE(same-named-attr)* is met quite easily when EXPRESS entities are represented as C++ classes, provided that the inheritance is declared virtual. On the same challenge CLOS does not map well; it combines the two attributes into one slot. However, by using the CLOS meta-object protocol, a relatively simple solution to this problem can be found.

Solutions to *CHALLENGE(redefined-attr)* require knowledge of the attribute's type in order to determine whether the type of the attribute of the child is indeed a restriction of the type found in the same-named attribute of the parent. The implementation language's ability to provide access to descriptive information about the class (*e.g,* names of attributes, attribute objects, superclasses, precedence ordering, *etc*.) can facilitate this effort. For example, in CLOS, user-defined classes are instances of **`standard-class`** or some programmer-defined subtype of it. **`standard-class`** is a subclass of **`standard-object`**, on which default behaviors for objects are defined. Since classes (*e.g.,* the classes representing EXPRESS entity types) are instances themselves, the same machinery used by other instances (*e.g.,* access to attribute values, object construction, *etc.*) can be reused by instances that are classes. An EXPRESS language environment can use this machinery directly, with one exception: since the type system and sense of type restriction defined in EXPRESS does not map directly to the Common Lisp type system, the programmer must define a method to order types by specificity.

In part because C++ classes are not objects themselves, C++ provides little support for solutions to this challenge. In C++, the **`typeid`** operation applied to an object can be used to obtain a **`type_info`** object for the original object, on which operations for type collating, type equality and access to the type's name are defined [X3J16]. However, no mechanism exists in C++ to identify the data members (*i.e.,* EXPRESS entity attributes) of a class. Therefore, some means other than querying the class must be found to obtain attributes. One method is to define 'dictionary' objects for both entities and attributes. These objects supplement the information found in the class with references to attribute objects. Dictionary objects are, in essence, a partial forfeiture from the mapping of EXPRESS entity types to classes.[6]

CHALLENGE(ANDOR-subtyping) is the problem of managing the proliferation of classes possible in ANDOR subtyping. That is, because of ANDOR subtyping, a large EXPRESS information model can result in the definition of hundreds of complex entity types. ANDOR subtyping, although an advantage from the modeler's standpoint, is a challenge to the language environment programmer. For the C++ programmer, the question is whether to implement each complex entity as a class, thereby increasing the

size of the image and compile time, or opting for some solution where complex entity types are not represented as a single C++ class, and thereby forfeiting some of the simplicity and elegance in the object model mapping. An effective and simple CLOS solution to the proliferation of classes was implemented in this work. It utilizes dynamic objects called 'programmatic classes' that are composed as needed from existing classes. [Kiczales]. This is described further in the next section.

3.0 ELEMENTS OF THE SOLUTION

The author is developing an EXPRESS language environment using Common Lisp and the CLOS metaobject protocol. A metaobject protocol is an interface to a language that gives the programmer the ability to incrementally modify the language's behavior and implementation [Kiczales]. The metaobject protocol is used to eliminate the mismatch between the implementation language (CLOS) object model and the target language (EXPRESS) object model so that the mechanisms of the implementation language's object system (*e.g.,* object definition, instance creation, method dispatching, *etc.*) can be reused. The motivation behind this approach is the belief that the reuse of these mechanisms will enhance the quality and speed of implementation.

The solution also employs dynamic object techniques (*e.g.*, run-time class definition, class redefinition). Dynamic object techniques support the requirements that the environment be responsive and that information models under development may be incrementally refined. Dynamic object techniques also provide a solution to the proliferation of classes that can occur because of EXPRESS ANDOR subtyping.

A principle benefit of implementing in Common Lisp is that the design may possess the advantages of both compiled and interpreted approaches with few of the disadvantages of either. For example, advantages of a compiled approach can be had: the system can translate EXPRESS information models into Common Lisp source code that can be compiled off-line and loaded at any time during a session with the system. Large libraries of existing EXPRESS information models can be accessed by the EXPRESS developer by these means. On the other hand, the benefits of an interpreter remain: EXPRESS written in the current session can immediately and incrementally be made available by run-time translation to Common Lisp and evaluation by the lisp interpreter. Because Common Lisp provides a resident programming environment and interpreter, the development of a complex interpreter 'execution engine' is obviated.

[6]The careful reader might recognize that, for even more fundamental reasons (because C++ does not allow run-time creation of classes) dictionary objects are necessary in every incremental development environment implemented in C++ (to describe entity types). The point here is to illustrate one advantage of having information about the class available in the class: the programmer has one less requirement on his implementation of dictionary objects.

In the system being developed, *CHALLENGE(ANDOR-subtyping)*, that is, the proliferation of classes that might result if all classes, including those that might never be populated, are created, is solved through the use of the programmatic class approach. A programmatic class, as described in [Kiczales], is a class generated at run-time by composing a from existing classes. The existing classes, in our case, are the classes representing simple entity types produced through translation of the EXPRESS source.

In an EXPRESS environment, generation of programmatic classes for the entity types requires knowledge of the *evaluated set*, the set of all legal entity types [ISO-11]. The evaluated set is calculated by a computation on the declaration of subtyping constraints (the *supertype-constraint clause*) of all entity declarations in the subject EXPRESS[7]. This computation generates a structure, **`complex-entity-type`**, for each element of the evaluated set. This structure identifies the simple entity types (those which are explicitly defined in the EXPRESS source) composing the complex entity type. When an EXPRESS entity instance is encountered through reading data in the exchange format of EXPRESS information models, [ISO-21], its type is checked against the evaluated set and, if legal and already generated, is instantiated. If the type is legal but the class has not yet been generated, the class is generated using the programmatic class mechanism. Through this means classes representing complex entity types are not defined until there is a need to instantiate an entity of that class. The programmatic class generated contains the information and attributes of each of the superclasses (simple entity types) from which it inherits.

CHALLENGE(same-named-attr) is a challenge for Common Lisp because CLOS classes inheriting a same-named slot from two superclasses produce a single slot. CLOS classes inherit this behavior from the standard class metaobject (sometimes called a *metaclass*) **`STANDARD-CLASS.`** The solution to this problem uses the metaobject protocol to provide a subclass of **`STANDARD-CLASS`** (called **`EXPRESS-CLASS`** here) that overrides the slot computation methods on **`STANDARD-CLASS`** with methods that produce multiple slots in accordance with EXPRESS object model behavior. Although the class of the classes representing EXPRESS entity types is no longer **`STANDARD-CLASS`**, other classes in the system are undisturbed by this modification.

The metaobject protocol is also used to ensure that all of the information found in an EXPRESS entity type definition can be encoded in the corresponding CLOS class object. For example, the standard CLOS metaobject defining slot features, **`STANDARD-SLOT`**, was subclassed by a class **`EXPRESS-SLOT`** to allow the recording of EXPRESS-specific features of attributes. For example, EXPRESS entity attributes may be declared as **`OPTIONAL`**, so that instances are free to leave that attribute's value unspecified. To implement this feature **`EXPRESS-SLOT`** extends the class **`STANDARD-SLOT`** with an additional slot, **`OPTIONAL-P`**, indi-

[7]The supertype-constraint clause includes the **`ANDOR`** constraint described earlier as well as **`AND`**, **`ONEOF`** and **`ABSTRACT`** constraints.

cating whether the EXPRESS attribute is optional. In a similar fashion the EXPRESS attribute features **DERIVED**, **INVERSE** and **UNIQUE**, and the recording of the attribute's source (the simple entity type from which it is inherited) and EXPRESS type is accommodated in the extended slot definition metaobject.

A solution to *CHALLENGE(redefined-attr)*, where it is necessary to determine whether the domain of an attribute in a subtype is indeed a restriction of the domain of the same-named attribute in the supertype, has yet to be implemented. The solution requires analysis of the types of both attributes and will probably be accomplished by unification[8] on the typed feature structures stored as type information of the attributes involved.

4.0 SYSTEM ARCHITECTURE

EXPRESS is a moderately large language.[9] Because of this, an efficient solution to the problem of translating EXPRESS source to Common Lisp forms is essential. The system, therefore, is comprised of three components:

- a core component that provides the behaviors of the EXPRESS object model, as described earlier, and entity instance reading and writing routines using STEP file-based exchange form, [ISO-21];
- an EXPRESS parser that builds typed feature structures from the input EXPRESS. (Feature structures are elements of a logic on which unification, generalization and specialization are defined). Typed feature structures are modeled as Common Lisp structures;
- a rule-based syntax transformation component that transforms the syntax tree of feature structures produced by the parser to a syntax tree of feature structures in the target language, Common Lisp.

The last two of these components are discussed in this section.

An EXPRESS parser is produced from the Zebu reversible LALR(1) parser generator [Laubsch]. Zebu generates parsers that are reversible in the sense that it produces code for both a "forward" parse to Common Lisp structures and a "reverse" parse back to the source language (*e.g.,* EXPRESS). The reverse parse is implemented as print functions on the Common Lisp structures generated in the forward parse. Hence when a Common Lisp structure resulting from the forward parse of EXPRESS is printed by Lisp, it appears as the original EXPRESS.[10]

Translation with the Zebu reversible parser can be performed by providing reversible parsers for both the input language (*e.g.,* EXPRESS) and the output language (*e.g.,* certain lisp forms). To perform the translation, the source language is forward parsed into structures, then rewrite rules are executed to transform the abstract syntax tree structures to a tree corresponding to the target language. Finally, the reverse parser (print functions) of the target language is used to produce the corresponding statements in the target language. Using this approach, translation is largely a matter of specifying the BNF for both languages and sets of rewrite rules to transform the syntax tree from the form of the source language to form of the target language.

The transformation of the abstract syntax tree is performed by a sister application of Zebu called Zebra, [Konrad]. Zebra provides a rule-based language for the pre-order and post-order transformation of abstract syntax trees produced by Zebu.

Zebu and Zebra have been very effective tools in overcoming the complexity of EXPRESS translation.

5.0 CONCLUSION

The immediate goal of this work has been the design of a tool for the efficient development of EXPRESS models. Such a tool will serve the needs of EXPRESS developers. The software should also find reuse in the development of EXPRESS translators (*e.g.,* to CORBA IDL [CORBA], Unified method [Booch], *etc.*). Beyond EXPRESS lay the larger aspects of the STEP architecture, (*e.g.,* application protocols, integrated resources). The application of these same techniques of meta-level and dynamic programming to these aspects STEP, for example, for the facilitation of reuse of integrated resources in application protocols, may provide an even greater benefit.

Finally, this work has shown that a good design of a language environment for a object flavored language may use the implementation language's metaobject protocol (where one exists) to align the implementation's object model behaviors with the needs of the target language objects. This approach may result in a smaller, more comprehensible program. Because the machinery of native object system is reused, the programmer may continue to use the familiar, native means of attribute access, object creation and method definition.

Dynamic object techniques are likewise critical to the solution. Dynamic object creation allows a straightforward solution to the proliferation of implicitly defined, potentially unpopulated classes that might otherwise result from EXPRESS ANDOR subtyping. Because classes may be created at run-time, the system can be far more responsive to the incremental development of the EXPRESS information model.

In addition to the techniques of dynamic object and meta-level programming, an efficient means to translate from the subject lan-

[8][Carpenter] provides an introduction to the logic of typed feature structures.

[9]The EXPRESS grammar consists of more than 450 productions.

[10]Recall that lisp programmers have the freedom to define how some lisp objects are printed by the lisp printer. Structures are printed according to a function defined in the **defstruct**. CLOS objects are printed according to the method **print-object**.

guage is necessary. Although, to date, only a small amount of necessary syntax transformation has been performed in this project, the combination of reversible parsers with rule-based tree transformation looks very promising.

ACKNOWLEDGEMENTS

The author would like to acknowledge the important contribution to the software development of this project made by Joachim Laubsch of Hewlett-Packard Laboratories, who developed Zebu and provided bug fixes and suggestions. Likewise, Karsten Konrad's program, Zebra, and his suggestions on its usage helped to make this work much easier. The author would also like to thank the workshop reviewers, David Terebessy and Lawrence Mayka and NIST reviewers, Bill Danner, KC Morris and Mary Mitchell for insightful comments. Bill Danner's ideas for future development are especially appreciated.

REFERENCES

[Booch] Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development*, Rational Software Corporation, 1995.

[Carpenter] Bob Carpenter, *The Logic of Typed Feature Structures*, Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, 1993

[CORBA] The Object Management Group, Inc., *The Common Object Request Broker: Architecture and Specification (CORBA),* The Object Management Group Inc., http://ruby.omg.org/pubodr.htm#Publications, 1995

[ISO-1] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Overview and Fundamental Principles*, International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step[11], 1994.

[ISO-11] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Description Methods: The EXPRESS Language Reference Manual*, International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-21] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Implementation Methods: Clear Text Encoding of the Exchange Structure,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-42] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Integrated Generic Resources: Geometric and Topological Representation,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-218] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Application Protocol: Ship Structures,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[Liang] Sheng Liang, Paul Hudak and Mark Jones, *Monad Transformers and Modular Interpreters*, Principles of Programming Languages'95, San Francisco, CA, January, 1995.

[Libes] Don Libes and Steve Clark, *The NIST EXPRESS Toolkit - Lessons Learned*, EXPRESS Users Group Conference Proceedings (EUG'92) Dallas, Texas, October 17-18, 1992.

[Kiczales] Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow, *The Art of the Metaobject Protocol,* The MIT Press, Cambridge, Massachusetts, 1991.

[Kiekenbeck] Juergen Kiekenbeck, Annette Siegenthaler, and Gunter Schlageter, *EXPRESS to C++: A mapping of the type-system*, EXPRESS Users Group Conference Proceedings (EUG'95) Grenoble, France, October 21-22, 1995.

[Konrad] Karsten Konrad, *Abstrakte Syntaxtransformation mit getypen Merkmalstermen*, Diplom Thesis, ftp://cl-ftp.dfki.uni-sb.de, September 23, 1994.

[Laubsch] Joachim Laubsch, *Zebu: A Tool for Specifying Reversible LALR(1) Parsers*. Hewlett-Packard Laboratories, Software Technology Laboratory, Internal Report, July 26, 1995. http://www.cs.cmu.edu/Web/Groups/AI/html/repository.html.

[Manola] Frank Manola, Editor, X3H7 Technical Committee (Object Information Management), *Features Matrix of Object Models,* Final Technical Report, ftp://ftp.gte.com/pub/dom/x3h7/adfinal.ps, March, 1996.

[Morris] Katherine C. Morris, David Sauder and Sandy Ressler, *Validation Testing System: Reusable Software Component Design*. National Institute of Standards and Technology, NISTIR 4937, October, 1992

[Schenck] Douglas Schenck and Peter Wilson, *Information Modeling: The EXPRESS Way*, Oxford University Press, 1994.

[STI] STEP Tools Inc., *Home Page*, http://www.steptools.com/index.html

[Wilson] Peter Wilson, *EXPRESS Tools and Services, (1990-1995)*, ftp://pub/step/express/tools/etools.ps, September, 1995.

[X3J16] ANSI Accredited Standards Committee X3J16, *Working Paper for Draft Proposed International Standard for Information Systems Programming Language C++*, http://www.cygnus.com/misc/wp/draft/, April 28, 1995.

[11] ISO IS level references are copyrighted and may be purchased from ISO. Draft copies can be obtained for free at this FTP address.

Dynamic Objects and Meta-level Programming of an EXPRESS Language Environment

Peter Denno

Manufacturing Systems Integration Division
National Institute of Standards and Technology
Gaithersburg, Maryland, USA

pdenno@cme.nist.gov

(301) 975-3595

ABSTRACT

This paper describes design and programming techniques employed in the development of a language environment for the EXPRESS information modeling language. A fundamental concern in the development of language environments for object flavored languages is the degree to which the object model of the implementation language matches that of the language being modeled. If there is significant mismatch, the programmer is forced to reconcile the differences with little aid from the implementation language's object model, *i.e.* eschewing the implementation's native object model and programming with elementary tools the object oriented behaviors of the target language environment. This paper describes how object model mismatch was eliminated and a responsive, incremental EXPRESS language environment is being developed using the Common Lisp Object System (CLOS) metaobject protocol (MOP) and dynamic object techniques.

1.0 INTRODUCTION

The paper begins with a presentation of background information regarding EXPRESS and its role in ISO 10303, more commonly referred to as STEP (The Standard for the Exchange of Product Model Data). Section 2.1 provides an overview of design principles commonly employed in the design of EXPRESS language tools. Sections 2.2 and 2.3 concern the problems of object model mismatch that confront designers of EXPRESS tools. Section 3.0 presents aspects of the author's Common Lisp implementation of an EXPRESS language environment. It describes solutions to the problems introduced in section 2 employing dynamic object techniques, CLOS and its metaobject protocol. Section 4.0 describes additional aspects of the EXPRESS language environment currently under development. The paper concludes with a summary of how dynamic object techniques and meta-level programming have contributed to the development of an EXPRESS language environment and how these same techniques might find use in the larger context of STEP development.

2.0 BACKGROUND

EXPRESS [ISO-11], [Schenck] is a formal language used to describe information models of STEP. The STEP standards support the unambiguous communication of industrial information exchanged in forms derived from EXPRESS language information models.[1] Within the STEP architecture, a *product model*, that is, an information model from which a range of similar artifacts can be described, is defined by an *application protocol* (for example, the application protocol, *Ship Structures*, [ISO-218]). Application protocols rely on libraries of commonly used concepts called *integrated resources*. The integrated resources collectively describe an abstract model of products. Components of the abstract model include, for example, *Geometric and Topological Representations* [ISO-42], which can be used in the description of application protocol for ship structures, automotive components, *etc.*. The reuse of common components improves the quality and development efficiency of application protocols and facilitates information sharing among related disciplines (*e.g.,* design systems and production scheduling systems may share an entity authorizing the release of a part for production).

Integrated resources and application protocols are defined in the EXPRESS language. Further discussion of the STEP architecture [ISO-1] is outside the scope of this paper.

[1] EXPRESS itself is an ISO standard developed by the STEP community.

In order to convey the information intended, systems that exchange data must agree upon the semantics of that data. A fundamental design goal of an information modeling language, such as EXPRESS, is to provide the means to describe information models (and subsequently data) reflecting a mutually understood semantics. That is, the language is used to define constraints which data sets exchanged must satisfy. In EXPRESS these constraints take several forms:

- **Constraints on an attribute's type[2] [3]:** the value of an attribute must be type-compatible with the type declared in the attributes's declaration.
- **Constraints on semantic consistency of entity instances:** Rules called *where rules* can be associated with an entity type. For example, an entity type named **`unit-vector`** possessing **`REAL`** valued attributes **`x`** and **`y`** might have a where rule requiring **`x**2 + y**2 = 1`**. These rules are procedurally defined in EXPRESS and may rely on built-in and user-defined functions and procedures.
- **Constraints on an entity instance's type:** EXPRESS provides a flexible mechanism for defining and composing hierarchical entity types (called *complex entity types*) and constraints on the legal composition of types within the hierarchy. For example, an abstract type **`person`** might have subtypes **`male`** and **`female`**. An instance of **`person`** must be one of **`male`** or **`female`** but not both.
- **Constraints on populations of entity instances:** Populations of entities (data sets) must satisfy constraints described in *global rules*. Such rules can, for example, constrain the cardinality of the instances of a type (*e.g.,* there is only one CEO) or require specific relationships between entities (*e.g.*, every employee has a supervisor).
- **Constraints on the existence of an entity instance:** The existence on an entity may depend on the existence of another entity.

2.1 Classification of EXPRESS Language Environments

A language environment for EXPRESS, that is, an environment that allows the development of EXPRESS information models and sample data sets, can be of great value to developers of EXPRESS information models. The EXPRESS for some STEP standards can run into the thousands of lines of code. Since the development of STEP standards has proved to be a lengthy process and a large part of that effort involves the articulation of an EXPRESS information model, the STEP community is quite interested in tools that improve the efficiency of EXPRESS development.

Several tools supporting EXPRESS development exist, [Wilson], [Libes], [Morris], [STI], [Kiekenbeck]. The focus of these tools is more the validation of data sets, the exploration of existing models, and the generation of application interfaces than it is the incremental development of new EXPRESS models. The work described here has as its goal the development of a responsive, incremental development environment for the concurrent development of EXPRESS models and representative data sets, (such an environment as is available to Lisp programmers).

The design of language environments include strictly interpreted approaches, strictly compiled approaches, and designs that utilize both interpretation and compilation. To date, compiled approaches to EXPRESS tools ([Morris], [STI], [Kiekenbeck]) use the subject EXPRESS to automatically generate code in an implementation language (*e.g.,* C++) which is then compiled and link-edited with auxiliary libraries that are independent from the subject code. Compiled approaches to EXPRESS environments have advantages over interpreted approaches in that libraries of compiled EXPRESS can be saved for later inclusion in information models without incurring the cost of interpretation. Also, the automatically generated source code is often useful in various applications that want access to data in the form defined by the subject EXPRESS model. Compiled approaches where compilation and link-edit are lengthy, however, can not support a responsive, incremental development environment: complete processing of large EXPRESS models using this approach can take 30 minutes or more.

A strictly interpreted approach can be more responsive to the incremental development of the EXPRESS subject code but can not take advantage of compiled libraries of supporting EXPRESS information models. Nor can interpreters implemented in statically compiled languages use the constructs of the language (*e.g.,* classes, iteration, arithmetic operations) in direct ways. For example, an interpreter can not generate and use a C++ class to represent an EXPRESS entity type. For these reasons, the development of an interpreter requires the development of fundamental supporting mechanisms unnecessary in compiled approaches. Therefore, the development of an interpreter for a language as large as EXPRESS is a daunting task. Recent developments in the design of interpreters, however, might make interpreted approaches more feasible, (*e.g.,* [Liang]).

Approaches that use both compilation and interpretation are commonly implemented in languages possessing resident programming environments, such as Smalltalk or Lisp.

2.2 Mismatch of Object Models

EXPRESS is an information modeling language, not a programming language. EXPRESS is 'object flavored' but not strictly object oriented. (It does not encapsulate state with methods). EXPRESS makes a distinction between classes and instances, allows class hierarchies and inheritance of data attributes.

An *object model* is a model with a class/instance distinction, encapsulation of state via methods and inheritance of behavior from

[2] *Attributes* correspond to C++ data members or CLOS slots. Attributes are associated with *entities*, analogous to C++ or CLOS classes.

[3] EXPRESS defines the usual primitive types and enumeration and allows user-defined types that generalize or specialize on primitive or other user-defined types

parent classes. *Concrete object models,* those underlying programming languages, vary with respect to inheritance, encapsulation, information hiding, dynamic capabilities and various other features [Manola]. A fundamental concern in the development of a language environment for an object flavored language is the degree to which the object model of the implementation language matches that of the language being modeled. If there is significant mismatch between the object model of the implementation language and that of the target language, the programmer is forced to reconcile the differences with little aid from the implementation language's object model. That is, software is not reused, but rather, beginning again with elementary tools, the programmer implements the object oriented behaviors of the target language environment. This, of course, is a great amount of work for what might be a small (but crucial!) deviation towards the behaviors of the target object model. This paper hopes to illustrate the value of a metaobject protocol (specifically the CLOS metaobject protocol) in reconciling the differences between the object models of the target and implementation languages.

2.3 Three Challenges in Object Model Mismatch[4]

The EXPRESS object model[5] differs in some important ways from the concrete object models underlying common programming languages (C++, Smalltalk, CLOS). Significant areas of mismatch between these languages and EXPRESS are found in the inheritance of attributes (C++ data members, or CLOS slots) and the approach to subclassing. Three characteristics of the EXPRESS language that present challenges in the development of an EXPRESS language environment are described below:

- ***CHALLENGE(same-named-attr):*** When same-named attributes of two EXPRESS entities are inherited through two distinct paths (EXPRESS allows multiple inheritance) two distinct attributes are inherited by the subclass, one from each parent defining the attribute.
- ***CHALLENGE(redefined-attr):*** In EXPRESS an attribute defined in a supertype may be redefined in the subtype only if the domain of values of the subtype is a restriction of the domain of the supertype. Thus an attribute of type **`number`** in the supertype can be redefined as type **`integer`** in the subtype, but an **`integer`** in the supertype can not be redefined as a **`number`** in the subtype.
- ***CHALLENGE(ANDOR-subtyping):*** Unless explicitly prohibited by a clause in the entity definition, the direct subtypes of an EXPRESS entity type can be combined to define additional subtypes.

[4]These are, of course, challenges for compiled approaches. Interpreted approaches using static implementation languages can not make use of the objects in these ways.

[5]Although not strictly object oriented, EXPRESS possesses enough of the characteristics of an object oriented language to make use of the term *object model* meaningful in this context.

An example of this behavior (called *ANDOR subtyping* here) is illustrated in the following example EXPRESS.

```
ENTITY a SUPERCLASS OF (b ANDOR c ANDOR d);
```

This clause declares that the entity combinations a, a+b, a+c, a+d, a+b+c, a+b+d, a+c+d and a+b+c+d are all instantiable, where a+b+c, for example, denotes an anonymous class (*complex entity type*) possessing the attributes (data members, slots) of the classes a, b and c.

Very different solutions to these challenges may be found depending on the implementation language chosen. *CHALLENGE(same-named-attr)* is met quite easily when EXPRESS entities are represented as C++ classes, provided that the inheritance is declared virtual. On the same challenge CLOS does not map well; it combines the two attributes into one slot. However, by using the CLOS meta-object protocol, a relatively simple solution to this problem can be found.

Solutions to *CHALLENGE(redefined-attr)* require knowledge of the attribute's type in order to determine whether the type of the attribute of the child is indeed a restriction of the type found in the same-named attribute of the parent. The implementation language's ability to provide access to descriptive information about the class (*e.g,* names of attributes, attribute objects, superclasses, precedence ordering, *etc*.) can facilitate this effort. For example, in CLOS, user-defined classes are instances of **`standard-class`** or some programmer-defined subtype of it. **`standard-class`** is a subclass of **`standard-object`**, on which default behaviors for objects are defined. Since classes (*e.g.,* the classes representing EXPRESS entity types) are instances themselves, the same machinery used by other instances (*e.g.,* access to attribute values, object construction, *etc.*) can be reused by instances that are classes. An EXPRESS language environment can use this machinery directly, with one exception: since the type system and sense of type restriction defined in EXPRESS does not map directly to the Common Lisp type system, the programmer must define a method to order types by specificity.

In part because C++ classes are not objects themselves, C++ provides little support for solutions to this challenge. In C++, the **`typeid`** operation applied to an object can be used to obtain a **`type_info`** object for the original object, on which operations for type collating, type equality and access to the type's name are defined [X3J16]. However, no mechanism exists in C++ to identify the data members (*i.e.,* EXPRESS entity attributes) of a class. Therefore, some means other than querying the class must be found to obtain attributes. One method is to define 'dictionary' objects for both entities and attributes. These objects supplement the information found in the class with references to attribute objects. Dictionary objects are, in essence, a partial forfeiture from the mapping of EXPRESS entity types to classes.[6]

CHALLENGE(ANDOR-subtyping) is the problem of managing the proliferation of classes possible in ANDOR subtyping. That is, because of ANDOR subtyping, a large EXPRESS information model can result in the definition of hundreds of complex entity types. ANDOR subtyping, although an advantage from the modeler's standpoint, is a challenge to the language environment programmer. For the C++ programmer, the question is whether to implement each complex entity as a class, thereby increasing the

size of the image and compile time, or opting for some solution where complex entity types are not represented as a single C++ class, and thereby forfeiting some of the simplicity and elegance in the object model mapping. An effective and simple CLOS solution to the proliferation of classes was implemented in this work. It utilizes dynamic objects called ‘programmatic classes’ that are composed as needed from existing classes. [Kiczales]. This is described further in the next section.

3.0 ELEMENTS OF THE SOLUTION

The author is developing an EXPRESS language environment using Common Lisp and the CLOS metaobject protocol. A metaobject protocol is an interface to a language that gives the programmer the ability to incrementally modify the language’s behavior and implementation [Kiczales]. The metaobject protocol is used to eliminate the mismatch between the implementation language (CLOS) object model and the target language (EXPRESS) object model so that the mechanisms of the implementation language’s object system (*e.g.,* object definition, instance creation, method dispatching, *etc.*) can be reused. The motivation behind this approach is the belief that the reuse of these mechanisms will enhance the quality and speed of implementation.

The solution also employs dynamic object techniques (*e.g.*, run-time class definition, class redefinition). Dynamic object techniques support the requirements that the environment be responsive and that information models under development may be incrementally refined. Dynamic object techniques also provide a solution to the proliferation of classes that can occur because of EXPRESS ANDOR subtyping.

A principle benefit of implementing in Common Lisp is that the design may possess the advantages of both compiled and interpreted approaches with few of the disadvantages of either. For example, advantages of a compiled approach can be had: the system can translate EXPRESS information models into Common Lisp source code that can be compiled off-line and loaded at any time during a session with the system. Large libraries of existing EXPRESS information models can be accessed by the EXPRESS developer by these means. On the other hand, the benefits of an interpreter remain: EXPRESS written in the current session can immediately and incrementally be made available by run-time translation to Common Lisp and evaluation by the lisp interpreter. Because Common Lisp provides a resident programming environment and interpreter, the development of a complex interpreter ‘execution engine’ is obviated.

[6]The careful reader might recognize that, for even more fundamental reasons (because C++ does not allow run-time creation of classes) dictionary objects are necessary in every incremental development environment implemented in C++ (to describe entity types). The point here is to illustrate one advantage of having information about the class available in the class: the programmer has one less requirement on his implementation of dictionary objects.

In the system being developed, *CHALLENGE(ANDOR-subtyping)*, that is, the proliferation of classes that might result if all classes, including those that might never be populated, are created, is solved through the use of the programmatic class approach. A programmatic class, as described in [Kiczales], is a class generated at run-time by composing a from existing classes. The existing classes, in our case, are the classes representing simple entity types produced through translation of the EXPRESS source.

In an EXPRESS environment, generation of programmatic classes for the entity types requires knowledge of the *evaluated set*, the set of all legal entity types [ISO-11]. The evaluated set is calculated by a computation on the declaration of subtyping constraints (the *supertype-constraint clause*) of all entity declarations in the subject EXPRESS[7]. This computation generates a structure, **`complex-entity-type`**, for each element of the evaluated set. This structure identifies the simple entity types (those which are explicitly defined in the EXPRESS source) composing the complex entity type. When an EXPRESS entity instance is encountered through reading data in the exchange format of EXPRESS information models, [ISO-21], its type is checked against the evaluated set and, if legal and already generated, is instantiated. If the type is legal but the class has not yet been generated, the class is generated using the programmatic class mechanism. Through this means classes representing complex entity types are not defined until there is a need to instantiate an entity of that class. The programmatic class generated contains the information and attributes of each of the superclasses (simple entity types) from which it inherits.

CHALLENGE(same-named-attr) is a challenge for Common Lisp because CLOS classes inheriting a same-named slot from two superclasses produce a single slot. CLOS classes inherit this behavior from the standard class metaobject (sometimes called a *metaclass*) **`STANDARD-CLASS.`** The solution to this problem uses the metaobject protocol to provide a subclass of **`STANDARD-CLASS`** (called **`EXPRESS-CLASS`** here) that overrides the slot computation methods on **`STANDARD-CLASS`** with methods that produce multiple slots in accordance with EXPRESS object model behavior. Although the class of the classes representing EXPRESS entity types is no longer **`STANDARD-CLASS`**, other classes in the system are undisturbed by this modification.

The metaobject protocol is also used to ensure that all of the information found in an EXPRESS entity type definition can be encoded in the corresponding CLOS class object. For example, the standard CLOS metaobject defining slot features, **`STANDARD-SLOT`**, was subclassed by a class **`EXPRESS-SLOT`** to allow the recording of EXPRESS-specific features of attributes. For example, EXPRESS entity attributes may be declared as **`OPTIONAL`**, so that instances are free to leave that attribute’s value unspecified. To implement this feature **`EXPRESS-SLOT`** extends the class **`STANDARD-SLOT`** with an additional slot, **`OPTIONAL-P`**, indi-

[7]The supertype-constraint clause includes the **`ANDOR`** constraint described earlier as well as **`AND`**, **`ONEOF`** and **`ABSTRACT`** constraints.

cating whether the EXPRESS attribute is optional. In a similar fashion the EXPRESS attribute features **DERIVED**, **INVERSE** and **UNIQUE**, and the recording of the attribute's source (the simple entity type from which it is inherited) and EXPRESS type is accommodated in the extended slot definition metaobject.

A solution to *CHALLENGE(redefined-attr)*, where it is necessary to determine whether the domain of an attribute in a subtype is indeed a restriction of the domain of the same-named attribute in the supertype, has yet to be implemented. The solution requires analysis of the types of both attributes and will probably be accomplished by unification[8] on the typed feature structures stored as type information of the attributes involved.

4.0 SYSTEM ARCHITECTURE

EXPRESS is a moderately large language.[9] Because of this, an efficient solution to the problem of translating EXPRESS source to Common Lisp forms is essential. The system, therefore, is comprised of three components:

- a core component that provides the behaviors of the EXPRESS object model, as described earlier, and entity instance reading and writing routines using STEP file-based exchange form, [ISO-21];
- an EXPRESS parser that builds typed feature structures from the input EXPRESS. (Feature structures are elements of a logic on which unification, generalization and specialization are defined). Typed feature structures are modeled as Common Lisp structures;
- a rule-based syntax transformation component that transforms the syntax tree of feature structures produced by the parser to a syntax tree of feature structures in the target language, Common Lisp.

The last two of these components are discussed in this section.

An EXPRESS parser is produced from the Zebu reversible LALR(1) parser generator [Laubsch]. Zebu generates parsers that are reversible in the sense that it produces code for both a "forward" parse to Common Lisp structures and a "reverse" parse back to the source language (*e.g.,* EXPRESS). The reverse parse is implemented as print functions on the Common Lisp structures generated in the forward parse. Hence when a Common Lisp structure resulting from the forward parse of EXPRESS is printed by Lisp, it appears as the original EXPRESS.[10]

Translation with the Zebu reversible parser can be performed by providing reversible parsers for both the input language (*e.g.,* EXPRESS) and the output language (*e.g.,* certain lisp forms). To perform the translation, the source language is forward parsed into structures, then rewrite rules are executed to transform the abstract syntax tree structures to a tree corresponding to the target language. Finally, the reverse parser (print functions) of the target language is used to produce the corresponding statements in the target language. Using this approach, translation is largely a matter of specifying the BNF for both languages and sets of rewrite rules to transform the syntax tree from the form of the source language to form of the target language.

The transformation of the abstract syntax tree is performed by a sister application of Zebu called Zebra, [Konrad]. Zebra provides a rule-based language for the pre-order and post-order transformation of abstract syntax trees produced by Zebu.

Zebu and Zebra have been very effective tools in overcoming the complexity of EXPRESS translation.

5.0 CONCLUSION

The immediate goal of this work has been the design of a tool for the efficient development of EXPRESS models. Such a tool will serve the needs of EXPRESS developers. The software should also find reuse in the development of EXPRESS translators (*e.g.,* to CORBA IDL [CORBA], Unified method [Booch], *etc.*). Beyond EXPRESS lay the larger aspects of the STEP architecture, (*e.g.,* application protocols, integrated resources). The application of these same techniques of meta-level and dynamic programming to these aspects STEP, for example, for the facilitation of reuse of integrated resources in application protocols, may provide an even greater benefit.

Finally, this work has shown that a good design of a language environment for a object flavored language may use the implementation language's metaobject protocol (where one exists) to align the implementation's object model behaviors with the needs of the target language objects. This approach may result in a smaller, more comprehensible program. Because the machinery of native object system is reused, the programmer may continue to use the familiar, native means of attribute access, object creation and method definition.

Dynamic object techniques are likewise critical to the solution. Dynamic object creation allows a straightforward solution to the proliferation of implicitly defined, potentially unpopulated classes that might otherwise result from EXPRESS ANDOR subtyping. Because classes may be created at run-time, the system can be far more responsive to the incremental development of the EXPRESS information model.

In addition to the techniques of dynamic object and meta-level programming, an efficient means to translate from the subject lan-

[8][Carpenter] provides an introduction to the logic of typed feature structures.

[9]The EXPRESS grammar consists of more than 450 productions.

[10]Recall that lisp programmers have the freedom to define how some lisp objects are printed by the lisp printer. Structures are printed according to a function defined in the **defstruct**. CLOS objects are printed according to the method **print-object**.

guage is necessary. Although, to date, only a small amount of necessary syntax transformation has been performed in this project, the combination of reversible parsers with rule-based tree transformation looks very promising.

ACKNOWLEDGEMENTS

The author would like to acknowledge the important contribution to the software development of this project made by Joachim Laubsch of Hewlett-Packard Laboratories, who developed Zebu and provided bug fixes and suggestions. Likewise, Karsten Konrad's program, Zebra, and his suggestions on its usage helped to make this work much easier. The author would also like to thank the workshop reviewers, David Terebessy and Lawrence Mayka and NIST reviewers, Bill Danner, KC Morris and Mary Mitchell for insightful comments. Bill Danner's ideas for future development are especially appreciated.

REFERENCES

[Booch] Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development*, Rational Software Corporation, 1995.

[Carpenter] Bob Carpenter, *The Logic of Typed Feature Structures*, Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, 1993

[CORBA] The Object Management Group, Inc., *The Common Object Request Broker: Architecture and Specification (CORBA),* The Object Management Group Inc., http://ruby.omg.org/pubodr.htm#Publications, 1995

[ISO-1] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Overview and Fundamental Principles*, International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step[11], 1994.

[ISO-11] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Description Methods: The EXPRESS Language Reference Manual*, International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-21] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Implementation Methods: Clear Text Encoding of the Exchange Structure,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-42] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Integrated Generic Resources: Geometric and Topological Representation,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-218] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Application Protocol: Ship Structures,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[Liang] Sheng Liang, Paul Hudak and Mark Jones, *Monad Transformers and Modular Interpreters*, Principles of Programming Languages'95, San Francisco, CA, January, 1995.

[Libes] Don Libes and Steve Clark, *The NIST EXPRESS Toolkit - Lessons Learned*, EXPRESS Users Group Conference Proceedings (EUG'92) Dallas, Texas, October 17-18, 1992.

[Kiczales] Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow, *The Art of the Metaobject Protocol,* The MIT Press, Cambridge, Massachusetts, 1991.

[Kiekenbeck] Juergen Kiekenbeck, Annette Siegenthaler, and Gunter Schlageter, *EXPRESS to C++: A mapping of the type-system*, EXPRESS Users Group Conference Proceedings (EUG'95) Grenoble, France, October 21-22, 1995.

[Konrad] Karsten Konrad, *Abstrakte Syntaxtransformation mit getypen Merkmalstermen*, Diplom Thesis, ftp://cl-ftp.dfki.uni-sb.de, September 23, 1994.

[Laubsch] Joachim Laubsch, *Zebu: A Tool for Specifying Reversible LALR(1) Parsers*. Hewlett-Packard Laboratories, Software Technology Laboratory, Internal Report, July 26, 1995. http://www.cs.cmu.edu/Web/Groups/AI/html/repository.html.

[Manola] Frank Manola, Editor, X3H7 Technical Committee (Object Information Management), *Features Matrix of Object Models,* Final Technical Report, ftp://ftp.gte.com/pub/dom/x3h7/adfinal.ps, March, 1996.

[Morris] Katherine C. Morris, David Sauder and Sandy Ressler, *Validation Testing System: Reusable Software Component Design*. National Institute of Standards and Technology, NISTIR 4937, October, 1992

[Schenck] Douglas Schenck and Peter Wilson, *Information Modeling: The EXPRESS Way*, Oxford University Press, 1994.

[STI] STEP Tools Inc., *Home Page*, http://www.steptools.com/index.html

[Wilson] Peter Wilson, *EXPRESS Tools and Services, (1990-1995)*, ftp://pub/step/express/tools/etools.ps, September, 1995.

[X3J16] ANSI Accredited Standards Committee X3J16, *Working Paper for Draft Proposed International Standard for Information Systems Programming Language C++*, http://www.cygnus.com/misc/wp/draft/, April 28, 1995.

[11] ISO IS level references are copyrighted and may be purchased from ISO. Draft copies can be obtained for free at this FTP address.

Dynamic Objects and Meta-level Programming of an EXPRESS Language Environment

Peter Denno

Manufacturing Systems Integration Division
National Institute of Standards and Technology
Gaithersburg, Maryland, USA

pdenno@cme.nist.gov

(301) 975-3595

ABSTRACT

This paper describes design and programming techniques employed in the development of a language environment for the EXPRESS information modeling language. A fundamental concern in the development of language environments for object flavored languages is the degree to which the object model of the implementation language matches that of the language being modeled. If there is significant mismatch, the programmer is forced to reconcile the differences with little aid from the implementation language's object model, *i.e.* eschewing the implementation's native object model and programming with elementary tools the object oriented behaviors of the target language environment. This paper describes how object model mismatch was eliminated and a responsive, incremental EXPRESS language environment is being developed using the Common Lisp Object System (CLOS) metaobject protocol (MOP) and dynamic object techniques.

1.0 INTRODUCTION

The paper begins with a presentation of background information regarding EXPRESS and its role in ISO 10303, more commonly referred to as STEP (The Standard for the Exchange of Product Model Data). Section 2.1 provides an overview of design principles commonly employed in the design of EXPRESS language tools. Sections 2.2 and 2.3 concern the problems of object model mismatch that confront designers of EXPRESS tools. Section 3.0 presents aspects of the author's Common Lisp implementation of an EXPRESS language environment. It describes solutions to the problems introduced in section 2 employing dynamic object techniques, CLOS and its metaobject protocol. Section 4.0 describes additional aspects of the EXPRESS language environment currently under development. The paper concludes with a summary of how dynamic object techniques and meta-level programming have contributed to the development of an EXPRESS language environment and how these same techniques might find use in the larger context of STEP development.

2.0 BACKGROUND

EXPRESS [ISO-11], [Schenck] is a formal language used to describe information models of STEP. The STEP standards support the unambiguous communication of industrial information exchanged in forms derived from EXPRESS language information models.[1] Within the STEP architecture, a *product model*, that is, an information model from which a range of similar artifacts can be described, is defined by an *application protocol* (for example, the application protocol, *Ship Structures*, [ISO-218]). Application protocols rely on libraries of commonly used concepts called *integrated resources*. The integrated resources collectively describe an abstract model of products. Components of the abstract model include, for example, *Geometric and Topological Representations* [ISO-42], which can be used in the description of application protocol for ship structures, automotive components, *etc.*. The reuse of common components improves the quality and development efficiency of application protocols and facilitates information sharing among related disciplines (*e.g.,* design systems and production scheduling systems may share an entity authorizing the release of a part for production).

Integrated resources and application protocols are defined in the EXPRESS language. Further discussion of the STEP architecture [ISO-1] is outside the scope of this paper.

[1] EXPRESS itself is an ISO standard developed by the STEP community.

In order to convey the information intended, systems that exchange data must agree upon the semantics of that data. A fundamental design goal of an information modeling language, such as EXPRESS, is to provide the means to describe information models (and subsequently data) reflecting a mutually understood semantics. That is, the language is used to define constraints which data sets exchanged must satisfy. In EXPRESS these constraints take several forms:

- **Constraints on an attribute's type**[2] [3]**:** the value of an attribute must be type-compatible with the type declared in the attributes's declaration.
- **Constraints on semantic consistency of entity instances:** Rules called *where rules* can be associated with an entity type. For example, an entity type named **`unit-vector`** possessing **`REAL`** valued attributes **`x`** and **`y`** might have a where rule requiring **`x**2 + y**2 = 1`**. These rules are procedurally defined in EXPRESS and may rely on built-in and user-defined functions and procedures.
- **Constraints on an entity instance's type:** EXPRESS provides a flexible mechanism for defining and composing hierarchical entity types (called *complex entity types*) and constraints on the legal composition of types within the hierarchy. For example, an abstract type **`person`** might have subtypes **`male`** and **`female`**. An instance of **`person`** must be one of **`male`** or **`female`** but not both.
- **Constraints on populations of entity instances:** Populations of entities (data sets) must satisfy constraints described in *global rules*. Such rules can, for example, constrain the cardinality of the instances of a type (*e.g.,* there is only one CEO) or require specific relationships between entities (*e.g.*, every employee has a supervisor).
- **Constraints on the existence of an entity instance:** The existence on an entity may depend on the existence of another entity.

2.1 Classification of EXPRESS Language Environments

A language environment for EXPRESS, that is, an environment that allows the development of EXPRESS information models and sample data sets, can be of great value to developers of EXPRESS information models. The EXPRESS for some STEP standards can run into the thousands of lines of code. Since the development of STEP standards has proved to be a lengthy process and a large part of that effort involves the articulation of an EXPRESS information model, the STEP community is quite interested in tools that improve the efficiency of EXPRESS development.

Several tools supporting EXPRESS development exist, [Wilson], [Libes], [Morris], [STI], [Kiekenbeck]. The focus of these tools is more the validation of data sets, the exploration of existing models, and the generation of application interfaces than it is the incremental development of new EXPRESS models. The work described here has as its goal the development of a responsive, incremental development environment for the concurrent development of EXPRESS models and representative data sets, (such an environment as is available to Lisp programmers).

The design of language environments include strictly interpreted approaches, strictly compiled approaches, and designs that utilize both interpretation and compilation. To date, compiled approaches to EXPRESS tools ([Morris], [STI], [Kiekenbeck]) use the subject EXPRESS to automatically generate code in an implementation language (*e.g.,* C++) which is then compiled and link-edited with auxiliary libraries that are independent from the subject code. Compiled approaches to EXPRESS environments have advantages over interpreted approaches in that libraries of compiled EXPRESS can be saved for later inclusion in information models without incurring the cost of interpretation. Also, the automatically generated source code is often useful in various applications that want access to data in the form defined by the subject EXPRESS model. Compiled approaches where compilation and link-edit are lengthy, however, can not support a responsive, incremental development environment: complete processing of large EXPRESS models using this approach can take 30 minutes or more.

A strictly interpreted approach can be more responsive to the incremental development of the EXPRESS subject code but can not take advantage of compiled libraries of supporting EXPRESS information models. Nor can interpreters implemented in statically compiled languages use the constructs of the language (*e.g.,* classes, iteration, arithmetic operations) in direct ways. For example, an interpreter can not generate and use a C++ class to represent an EXPRESS entity type. For these reasons, the development of an interpreter requires the development of fundamental supporting mechanisms unnecessary in compiled approaches. Therefore, the development of an interpreter for a language as large as EXPRESS is a daunting task. Recent developments in the design of interpreters, however, might make interpreted approaches more feasible, (*e.g.,* [Liang]).

Approaches that use both compilation and interpretation are commonly implemented in languages possessing resident programming environments, such as Smalltalk or Lisp.

2.2 Mismatch of Object Models

EXPRESS is an information modeling language, not a programming language. EXPRESS is 'object flavored' but not strictly object oriented. (It does not encapsulate state with methods). EXPRESS makes a distinction between classes and instances, allows class hierarchies and inheritance of data attributes.

An *object model* is a model with a class/instance distinction, encapsulation of state via methods and inheritance of behavior from

[2]*Attributes* correspond to C++ data members or CLOS slots. Attributes are associated with *entities*, analogous to C++ or CLOS classes.

[3]EXPRESS defines the usual primitive types and enumeration and allows user-defined types that generalize or specialize on primitive or other user-defined types

parent classes. *Concrete object models,* those underlying programming languages, vary with respect to inheritance, encapsulation, information hiding, dynamic capabilities and various other features [Manola]. A fundamental concern in the development of a language environment for an object flavored language is the degree to which the object model of the implementation language matches that of the language being modeled. If there is significant mismatch between the object model of the implementation language and that of the target language, the programmer is forced to reconcile the differences with little aid from the implementation language's object model. That is, software is not reused, but rather, beginning again with elementary tools, the programmer implements the object oriented behaviors of the target language environment. This, of course, is a great amount of work for what might be a small (but crucial!) deviation towards the behaviors of the target object model. This paper hopes to illustrate the value of a metaobject protocol (specifically the CLOS metaobject protocol) in reconciling the differences between the object models of the target and implementation languages.

2.3 Three Challenges in Object Model Mismatch[4]

The EXPRESS object model[5] differs in some important ways from the concrete object models underlying common programming languages (C++, Smalltalk, CLOS). Significant areas of mismatch between these languages and EXPRESS are found in the inheritance of attributes (C++ data members, or CLOS slots) and the approach to subclassing. Three characteristics of the EXPRESS language that present challenges in the development of an EXPRESS language environment are described below:

- ***CHALLENGE(same-named-attr):*** When same-named attributes of two EXPRESS entities are inherited through two distinct paths (EXPRESS allows multiple inheritance) two distinct attributes are inherited by the subclass, one from each parent defining the attribute.
- ***CHALLENGE(redefined-attr):*** In EXPRESS an attribute defined in a supertype may be redefined in the subtype only if the domain of values of the subtype is a restriction of the domain of the supertype. Thus an attribute of type **`number`** in the supertype can be redefined as type **`integer`** in the subtype, but an **`integer`** in the supertype can not be redefined as a **`number`** in the subtype.
- ***CHALLENGE(ANDOR-subtyping):*** Unless explicitly prohibited by a clause in the entity definition, the direct subtypes of an EXPRESS entity type can be combined to define additional subtypes.

[4]These are, of course, challenges for compiled approaches. Interpreted approaches using static implementation languages can not make use of the objects in these ways.

[5]Although not strictly object oriented, EXPRESS possesses enough of the characteristics of an object oriented language to make use of the term *object model* meaningful in this context.

An example of this behavior (called *ANDOR subtyping* here) is illustrated in the following example EXPRESS.

```
ENTITY a SUPERCLASS OF (b ANDOR c ANDOR d);
```

This clause declares that the entity combinations a, a+b, a+c, a+d, a+b+c, a+b+d, a+c+d and a+b+c+d are all instantiable, where a+b+c, for example, denotes an anonymous class (*complex entity type*) possessing the attributes (data members, slots) of the classes a, b and c.

Very different solutions to these challenges may be found depending on the implementation language chosen. *CHALLENGE(same-named-attr)* is met quite easily when EXPRESS entities are represented as C++ classes, provided that the inheritance is declared virtual. On the same challenge CLOS does not map well; it combines the two attributes into one slot. However, by using the CLOS meta-object protocol, a relatively simple solution to this problem can be found.

Solutions to *CHALLENGE(redefined-attr)* require knowledge of the attribute's type in order to determine whether the type of the attribute of the child is indeed a restriction of the type found in the same-named attribute of the parent. The implementation language's ability to provide access to descriptive information about the class (*e.g,* names of attributes, attribute objects, superclasses, precedence ordering, *etc*.) can facilitate this effort. For example, in CLOS, user-defined classes are instances of **`standard-class`** or some programmer-defined subtype of it. **`standard-class`** is a subclass of **`standard-object`**, on which default behaviors for objects are defined. Since classes (*e.g.,* the classes representing EXPRESS entity types) are instances themselves, the same machinery used by other instances (*e.g.,* access to attribute values, object construction, *etc.*) can be reused by instances that are classes. An EXPRESS language environment can use this machinery directly, with one exception: since the type system and sense of type restriction defined in EXPRESS does not map directly to the Common Lisp type system, the programmer must define a method to order types by specificity.

In part because C++ classes are not objects themselves, C++ provides little support for solutions to this challenge. In C++, the **`typeid`** operation applied to an object can be used to obtain a **`type_info`** object for the original object, on which operations for type collating, type equality and access to the type's name are defined [X3J16]. However, no mechanism exists in C++ to identify the data members (*i.e.,* EXPRESS entity attributes) of a class. Therefore, some means other than querying the class must be found to obtain attributes. One method is to define 'dictionary' objects for both entities and attributes. These objects supplement the information found in the class with references to attribute objects. Dictionary objects are, in essence, a partial forfeiture from the mapping of EXPRESS entity types to classes.[6]

CHALLENGE(ANDOR-subtyping) is the problem of managing the proliferation of classes possible in ANDOR subtyping. That is, because of ANDOR subtyping, a large EXPRESS information model can result in the definition of hundreds of complex entity types. ANDOR subtyping, although an advantage from the modeler's standpoint, is a challenge to the language environment programmer. For the C++ programmer, the question is whether to implement each complex entity as a class, thereby increasing the

size of the image and compile time, or opting for some solution where complex entity types are not represented as a single C++ class, and thereby forfeiting some of the simplicity and elegance in the object model mapping. An effective and simple CLOS solution to the proliferation of classes was implemented in this work. It utilizes dynamic objects called 'programmatic classes' that are composed as needed from existing classes. [Kiczales]. This is described further in the next section.

3.0 ELEMENTS OF THE SOLUTION

The author is developing an EXPRESS language environment using Common Lisp and the CLOS metaobject protocol. A metaobject protocol is an interface to a language that gives the programmer the ability to incrementally modify the language's behavior and implementation [Kiczales]. The metaobject protocol is used to eliminate the mismatch between the implementation language (CLOS) object model and the target language (EXPRESS) object model so that the mechanisms of the implementation language's object system (*e.g.,* object definition, instance creation, method dispatching, *etc.*) can be reused. The motivation behind this approach is the belief that the reuse of these mechanisms will enhance the quality and speed of implementation.

The solution also employs dynamic object techniques (*e.g.*, run-time class definition, class redefinition). Dynamic object techniques support the requirements that the environment be responsive and that information models under development may be incrementally refined. Dynamic object techniques also provide a solution to the proliferation of classes that can occur because of EXPRESS ANDOR subtyping.

A principle benefit of implementing in Common Lisp is that the design may possess the advantages of both compiled and interpreted approaches with few of the disadvantages of either. For example, advantages of a compiled approach can be had: the system can translate EXPRESS information models into Common Lisp source code that can be compiled off-line and loaded at any time during a session with the system. Large libraries of existing EXPRESS information models can be accessed by the EXPRESS developer by these means. On the other hand, the benefits of an interpreter remain: EXPRESS written in the current session can immediately and incrementally be made available by run-time translation to Common Lisp and evaluation by the lisp interpreter. Because Common Lisp provides a resident programming environment and interpreter, the development of a complex interpreter 'execution engine' is obviated.

[6]The careful reader might recognize that, for even more fundamental reasons (because C++ does not allow run-time creation of classes) dictionary objects are necessary in every incremental development environment implemented in C++ (to describe entity types). The point here is to illustrate one advantage of having information about the class available in the class: the programmer has one less requirement on his implementation of dictionary objects.

In the system being developed, *CHALLENGE(ANDOR-subtyping)*, that is, the proliferation of classes that might result if all classes, including those that might never be populated, are created, is solved through the use of the programmatic class approach. A programmatic class, as described in [Kiczales], is a class generated at run-time by composing a from existing classes. The existing classes, in our case, are the classes representing simple entity types produced through translation of the EXPRESS source.

In an EXPRESS environment, generation of programmatic classes for the entity types requires knowledge of the *evaluated set*, the set of all legal entity types [ISO-11]. The evaluated set is calculated by a computation on the declaration of subtyping constraints (the *supertype-constraint clause*) of all entity declarations in the subject EXPRESS[7]. This computation generates a structure, **`complex-entity-type`**, for each element of the evaluated set. This structure identifies the simple entity types (those which are explicitly defined in the EXPRESS source) composing the complex entity type. When an EXPRESS entity instance is encountered through reading data in the exchange format of EXPRESS information models, [ISO-21], its type is checked against the evaluated set and, if legal and already generated, is instantiated. If the type is legal but the class has not yet been generated, the class is generated using the programmatic class mechanism. Through this means classes representing complex entity types are not defined until there is a need to instantiate an entity of that class. The programmatic class generated contains the information and attributes of each of the superclasses (simple entity types) from which it inherits.

CHALLENGE(same-named-attr) is a challenge for Common Lisp because CLOS classes inheriting a same-named slot from two superclasses produce a single slot. CLOS classes inherit this behavior from the standard class metaobject (sometimes called a *metaclass*) **`STANDARD-CLASS.`** The solution to this problem uses the metaobject protocol to provide a subclass of **`STANDARD-CLASS`** (called **`EXPRESS-CLASS`** here) that overrides the slot computation methods on **`STANDARD-CLASS`** with methods that produce multiple slots in accordance with EXPRESS object model behavior. Although the class of the classes representing EXPRESS entity types is no longer **`STANDARD-CLASS`**, other classes in the system are undisturbed by this modification.

The metaobject protocol is also used to ensure that all of the information found in an EXPRESS entity type definition can be encoded in the corresponding CLOS class object. For example, the standard CLOS metaobject defining slot features, **`STANDARD-SLOT`**, was subclassed by a class **`EXPRESS-SLOT`** to allow the recording of EXPRESS-specific features of attributes. For example, EXPRESS entity attributes may be declared as **`OPTIONAL`**, so that instances are free to leave that attribute's value unspecified. To implement this feature **`EXPRESS-SLOT`** extends the class **`STANDARD-SLOT`** with an additional slot, **`OPTIONAL-P`**, indi-

[7]The supertype-constraint clause includes the **`ANDOR`** constraint described earlier as well as **`AND`**, **`ONEOF`** and **`ABSTRACT`** constraints.

cating whether the EXPRESS attribute is optional. In a similar fashion the EXPRESS attribute features **DERIVED**, **INVERSE** and **UNIQUE**, and the recording of the attribute's source (the simple entity type from which it is inherited) and EXPRESS type is accommodated in the extended slot definition metaobject.

A solution to *CHALLENGE(redefined-attr)*, where it is necessary to determine whether the domain of an attribute in a subtype is indeed a restriction of the domain of the same-named attribute in the supertype, has yet to be implemented. The solution requires analysis of the types of both attributes and will probably be accomplished by unification[8] on the typed feature structures stored as type information of the attributes involved.

4.0 SYSTEM ARCHITECTURE

EXPRESS is a moderately large language.[9] Because of this, an efficient solution to the problem of translating EXPRESS source to Common Lisp forms is essential. The system, therefore, is comprised of three components:

- a core component that provides the behaviors of the EXPRESS object model, as described earlier, and entity instance reading and writing routines using STEP file-based exchange form, [ISO-21];
- an EXPRESS parser that builds typed feature structures from the input EXPRESS. (Feature structures are elements of a logic on which unification, generalization and specialization are defined). Typed feature structures are modeled as Common Lisp structures;
- a rule-based syntax transformation component that transforms the syntax tree of feature structures produced by the parser to a syntax tree of feature structures in the target language, Common Lisp.

The last two of these components are discussed in this section.

An EXPRESS parser is produced from the Zebu reversible LALR(1) parser generator [Laubsch]. Zebu generates parsers that are reversible in the sense that it produces code for both a "forward" parse to Common Lisp structures and a "reverse" parse back to the source language (*e.g.,* EXPRESS). The reverse parse is implemented as print functions on the Common Lisp structures generated in the forward parse. Hence when a Common Lisp structure resulting from the forward parse of EXPRESS is printed by Lisp, it appears as the original EXPRESS.[10]

Translation with the Zebu reversible parser can be performed by providing reversible parsers for both the input language (*e.g.,* EXPRESS) and the output language (*e.g.,* certain lisp forms). To perform the translation, the source language is forward parsed into structures, then rewrite rules are executed to transform the abstract syntax tree structures to a tree corresponding to the target language. Finally, the reverse parser (print functions) of the target language is used to produce the corresponding statements in the target language. Using this approach, translation is largely a matter of specifying the BNF for both languages and sets of rewrite rules to transform the syntax tree from the form of the source language to form of the target language.

The transformation of the abstract syntax tree is performed by a sister application of Zebu called Zebra, [Konrad]. Zebra provides a rule-based language for the pre-order and post-order transformation of abstract syntax trees produced by Zebu.

Zebu and Zebra have been very effective tools in overcoming the complexity of EXPRESS translation.

5.0 CONCLUSION

The immediate goal of this work has been the design of a tool for the efficient development of EXPRESS models. Such a tool will serve the needs of EXPRESS developers. The software should also find reuse in the development of EXPRESS translators (*e.g.,* to CORBA IDL [CORBA], Unified method [Booch], *etc.*). Beyond EXPRESS lay the larger aspects of the STEP architecture, (*e.g.,* application protocols, integrated resources). The application of these same techniques of meta-level and dynamic programming to these aspects STEP, for example, for the facilitation of reuse of integrated resources in application protocols, may provide an even greater benefit.

Finally, this work has shown that a good design of a language environment for a object flavored language may use the implementation language's metaobject protocol (where one exists) to align the implementation's object model behaviors with the needs of the target language objects. This approach may result in a smaller, more comprehensible program. Because the machinery of native object system is reused, the programmer may continue to use the familiar, native means of attribute access, object creation and method definition.

Dynamic object techniques are likewise critical to the solution. Dynamic object creation allows a straightforward solution to the proliferation of implicitly defined, potentially unpopulated classes that might otherwise result from EXPRESS ANDOR subtyping. Because classes may be created at run-time, the system can be far more responsive to the incremental development of the EXPRESS information model.

In addition to the techniques of dynamic object and meta-level programming, an efficient means to translate from the subject lan-

[8][Carpenter] provides an introduction to the logic of typed feature structures.

[9]The EXPRESS grammar consists of more than 450 productions.

[10]Recall that lisp programmers have the freedom to define how some lisp objects are printed by the lisp printer. Structures are printed according to a function defined in the `defstruct`. CLOS objects are printed according to the method `print-object`.

guage is necessary. Although, to date, only a small amount of necessary syntax transformation has been performed in this project, the combination of reversible parsers with rule-based tree transformation looks very promising.

ACKNOWLEDGEMENTS

The author would like to acknowledge the important contribution to the software development of this project made by Joachim Laubsch of Hewlett-Packard Laboratories, who developed Zebu and provided bug fixes and suggestions. Likewise, Karsten Konrad's program, Zebra, and his suggestions on its usage helped to make this work much easier. The author would also like to thank the workshop reviewers, David Terebessy and Lawrence Mayka and NIST reviewers, Bill Danner, KC Morris and Mary Mitchell for insightful comments. Bill Danner's ideas for future development are especially appreciated.

REFERENCES

[Booch] Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development*, Rational Software Corporation, 1995.

[Carpenter] Bob Carpenter, *The Logic of Typed Feature Structures*, Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, 1993

[CORBA] The Object Management Group, Inc., *The Common Object Request Broker: Architecture and Specification (CORBA),* The Object Management Group Inc., http://ruby.omg.org/pubodr.htm#Publications, 1995

[ISO-1] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Overview and Fundamental Principles*, International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step[11], 1994.

[ISO-11] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Description Methods: The EXPRESS Language Reference Manual*, International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-21] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Implementation Methods: Clear Text Encoding of the Exchange Structure,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-42] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Integrated Generic Resources: Geometric and Topological Representation,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[ISO-218] International Organization for Standardization, *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Application Protocol: Ship Structures,* International Standard, ISO TC184/SC4, ftp://ftp.cme.nist.gov/pub/step, 1994.

[Liang] Sheng Liang, Paul Hudak and Mark Jones, *Monad Transformers and Modular Interpreters*, Principles of Programming Languages'95, San Francisco, CA, January, 1995.

[Libes] Don Libes and Steve Clark, *The NIST EXPRESS Toolkit - Lessons Learned*, EXPRESS Users Group Conference Proceedings (EUG'92) Dallas, Texas, October 17-18, 1992.

[Kiczales] Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow, *The Art of the Metaobject Protocol,* The MIT Press, Cambridge, Massachusetts, 1991.

[Kiekenbeck] Juergen Kiekenbeck, Annette Siegenthaler, and Gunter Schlageter, *EXPRESS to C++: A mapping of the type-system*, EXPRESS Users Group Conference Proceedings (EUG'95) Grenoble, France, October 21-22, 1995.

[Konrad] Karsten Konrad, *Abstrakte Syntaxtransformation mit getypen Merkmalstermen*, Diplom Thesis, ftp://cl-ftp.dfki.uni-sb.de, September 23, 1994.

[Laubsch] Joachim Laubsch, *Zebu: A Tool for Specifying Reversible LALR(1) Parsers*. Hewlett-Packard Laboratories, Software Technology Laboratory, Internal Report, July 26, 1995. http://www.cs.cmu.edu/Web/Groups/AI/html/repository.html.

[Manola] Frank Manola, Editor, X3H7 Technical Committee (Object Information Management), *Features Matrix of Object Models,* Final Technical Report, ftp://ftp.gte.com/pub/dom/x3h7/adfinal.ps, March, 1996.

[Morris] Katherine C. Morris, David Sauder and Sandy Ressler, *Validation Testing System: Reusable Software Component Design*. National Institute of Standards and Technology, NISTIR 4937, October, 1992

[Schenck] Douglas Schenck and Peter Wilson, *Information Modeling: The EXPRESS Way*, Oxford University Press, 1994.

[STI] STEP Tools Inc., *Home Page*, http://www.steptools.com/index.html

[Wilson] Peter Wilson, *EXPRESS Tools and Services, (1990-1995)*, ftp://pub/step/express/tools/etools.ps, September, 1995.

[X3J16] ANSI Accredited Standards Committee X3J16, *Working Paper for Draft Proposed International Standard for Information Systems Programming Language C++*, http://www.cygnus.com/misc/wp/draft/, April 28, 1995.

[11] ISO IS level references are copyrighted and may be purchased from ISO. Draft copies can be obtained for free at this FTP address.