An Example of Object Oriented Design Recovery: Modula-3 Interfaces for ATK

Preliminary Report
June 22, 1992

Richard J Orgass*
Joseph M Newcomer**
David B Anderson

Information Technology Center
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania USA

Abstract

The Andrew Toolkit (ATK) is an object oriented C library and applications for the construction and manipulation of mixed media documents and mail that predates the widespread use of the currently popular object oriented preprocessors for C such as C++,[1] Eifel[2] and Objective-C.[3] A preprocessor, *class*, translates casually (partially?) typed source text that consists of class declarations and assorted C declarations into two include files, one for the implementor of the class and one for the clients of the class.

This paper describes our experiences providing Modula-3[4] interfaces to existing C code for ATK by providing a modified version of the class preprocessor and modified source for the preprocessor so that Modula-3 interfaces can be generated automatically and replacing the existing ATK runtime environment with the DEC SRC Modula-3 runtime environment using shared libraries in place of dynamic loading.

As a result of this work, ATK applications written in Modula-3 and C can be used together and, over time, ATK code written in pre-ANSI C can be converted to Modula-3 without disturbing client applications written in either language. It is, however, necessary to change the text of the class source files to provide missing semantic information.

The most surprising result is that the above can be achieved with quite ordinary and routine methods and tools.

---

* Present address: IBM Industrial Technology Center, 4 Allegheny Center, Pittsburgh, PA 15231.

** Present address: Joseph M Newcomer Co, 610 Kirtland Street, Pittsburgh, PA 15208.

[1] B Stroustrop. *The C++ Programming Language.* Reading, Massachusetts, Addison-Wesley, 1986.

[2] B Meyer. *Object-Oriented Software Construction.* Englewood Cliffs, NJ, Prentice-Hall, 1988.

[3] B J Cox. *Object Oriented Programming: An Evolutionary Approach.* Englewood Clifs, NJ, Prentice-Hall, 1986.

[4] G Nelson (Ed). *Systems Programming with Modula-3.* Englewood Cliffs, NJ, Prentice-Hall , 1991.

1. Introduction

This section contains a brief description of the Andrew Toolkit, a description of our motivation for undertaking the work described in this paper and a statement of the problem that was to be solved.

1.1 Andrew Toolkit

Andrew Toolkit (ATK) is an object oriented toolkit for the development of embedded documents with single parent inheritance. These documents are active objects. The available insets (objects that can be embedded in documents) include multi-font formatted text, raster images, spread sheets, line and shading drawings, annimations, etc. The toolkit is extended by adding new insets to the library. The disk file representation of ATK documents (Andrew data streams) consists of printable ASCII characters and is formatted so that the documents can be transmitted via electronic mail.

The two main ATK applications are the editor *ez* and *messages*, a user interface to the Andrew Mail System (AMS). The editor ez allows users to display and modify both plain text and ATK embedded documents. The program messages makes it possible for users to perform all of the usual operations on electronic mail. In addition, users my file messages into folders, retrieve mail from folders, subscribe to folders, etc. A folder that contains mail sent to a single mailing list or address is usually referred to as a bulletin board.

The toolkit was developed as a research project that began in 1984, before C++ and other preprocessors for C were available. One of the objectives was to demonstrate that it is possible to construct such documents. The toolkit is in production use in several locations. It consists of about 250 classes and about 300,000 lines of source code.

The object types or classes of ATK are declared using a toolkit specific language and preprocessor called *class*. The result of running the program `class` against a declaration of an ATK class (in a `.ch` file) is two C include files, one for the C compilation units that implement the object (a `.eh` file) and one for the C compilation units that are clients of the object declaration and implementation (a `.ih` file). The contents of the emitted include files are primarily C preprocessor macros for the invocation of methods and other procedures as well as a variety of other runtime administrative data and code. Almost all type information is optional and treated as commentary in `.ch` files.

1.2 Motivation

As part of an evaluation of Modula-3 as a supported language for product software development, a number of single person test cases were successfully completed. At the next step in the evaluation, Modula-3 was to be used for a committed internal or external product with the property that the penalty for failure would be clearly non-trivial but acceptable. Based on available Modula-3 compilers and expected projects it was determined that the first pilot project would be done with the Andrew Toolkit (ATK) provided that Modula-3 interfaces could be produced at an acceptable cost.

Both Modula-3 and ATK provide objects with single parent inheritance and an object model that is based on reasonably similar views of objects, namely, data records and method records. In spite of a number of differences of detail, the heavy use of C macros in the ATK code lead us to believe that it would be possible to mechanically convert an overwhelming majority of the ATK object class declaration code (`.ch` files) to Modula-3. All of the authors initially viewed the project as a piece of routine but complicated programming that is not particularly interesting.

The successful completion of the work required recovering, checking and correcting type information that was not recorded when ATK was implemented. Indeed, there are parts of the code that will not work in the presence of type checking at the level of ANSI C. A significantly enhanced class preprocessor was needed to deal with the recovered information. In addition, a small enhancement to generate Modula-3 interfaces was required.

1.3 Problem Statement

This paper describes the work that was done to provide a set of Modula-3 libraries that declare and implement essentially all of the classes of the Andrew Toolkit in such a way that:

(1) The ATK design and pre-ANSI C implementation is preserved. That is, the existing C code is changed only in such a way that exactly the same source (without conditional compilation) can be used to implement both the Modula-3 interfaces and the conventional C build. Two sets of library binaries are acceptable.

(2) The class declaration files may be changed to support translation to Modula-3 provided that exactly the same `.ch` files can be used to build ATK from C.

(3) The type information added to the class declaration files is to also appear in the generated C header files so that much more type checking can be done in the C source.

(4) Shared libraries are to be used in place of the much older dynamic linking used in current versions of ATK.

The conclusion of the paper describes some principles and techniques that may be relevant to other object-oriented design recovery activities.

2. Object Models

This background section describes the object models used by the traditional ATK environment and the DEC SRC Modula-3 object model. The material presented here will be used in later sections. Readers who are familiar with these object models may wish to skip the section.

2.1 ATK Object Model

Class procedures are associated with ATK objects. These procedures may be divided into two groups: (1) Procedures that provide for the initialization and/or creation of object instances or for the deallocation of instance dynamic data and/or the destruction of object instances. (2) Class procedures that make it possible for a client to change the shared state of all instances of the class.

In many other object oriented languages these functions are provided by language primitives and procedures that are declared in the interface or include file that declares the class. ATK uses a local implementation of dynamic loading that was designed to provide this function for BSD 4.2 and 4.3 systems and has been carried forward to modern operating systems. Both the method and class procedure code is dynamically loaded so pointers to the appropriate code must be part of each object instance so that the code can be found at run time.

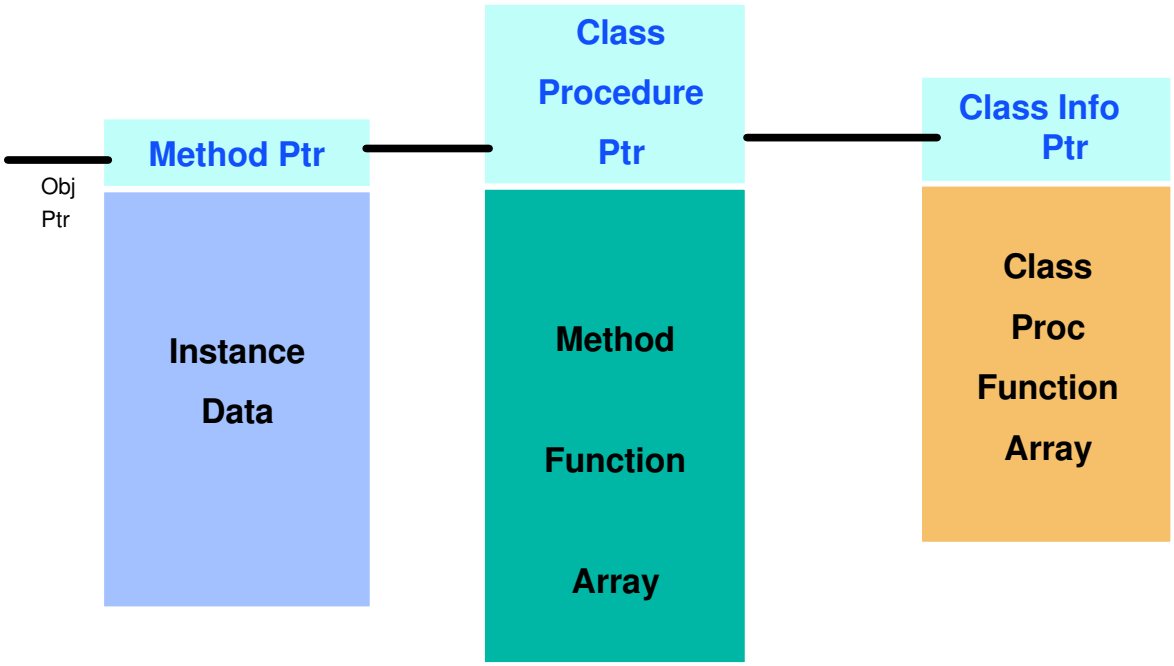

Figure 1. Traditional ATK Object Model

A traditional ATK object instance (Figure 1) can be divided into two parts. The first part, which is the target of the object pointer, contains a pointer to the class specific data and code and the instance specific data.

The destination of an ATK object pointer is a record (struct for C programmers) with two fields: The first field is a pointer to the class specific data and the second field is a record containing the instance specific data of the object.

The pointer to the class specific data is actually a pointer to another record with two fields. The first field is a pointer to class information and class procedures and the second is an array of pointers to functions that implement the methods of an object of this type.

For this work, the target of the pointer to class information and procedures may be viewed as a record with several fields. The first field is a unique unsigned 32 bit integer that serves as a unique identifier for objects of this type, that is, a type code. The remaining fields are pointers to functions that implement the class procedures in the class declaration.

Access to the data. methods and class procedures of an object are provided to client and implementor authors by C macros that are generated by the class preprocessor (in the output `.ih` and `.eh` files). The fact that C macros are used for this purpose makes it possible to use the same C source without conditional compilation for the implementation of both the C and Modula-3 version of ATK.

2.2 Modula-3 Object Model

Modula-3 includes the concepts of traced and untraced references and objects. Both reference types and objects are implemented as pointers. For traced objects and reference types, the Modula-3 compiler and runtime provide garbage collection and guarantee the validity of pointers. In contrast, for untraced object and reference types it is the programmer that guarantees both the validity of pointers and that storage is released when it is no longer needed.

The referent of each traced reference and each object (traced or untraced) has the property that the word before the target of the pointer contains an integer called a type code. In each linked module, every traced reference type and every object type (traced or untraced) is assigned a unique type code that uniquely identifies this type as well as all known runtime information about instances of the type.

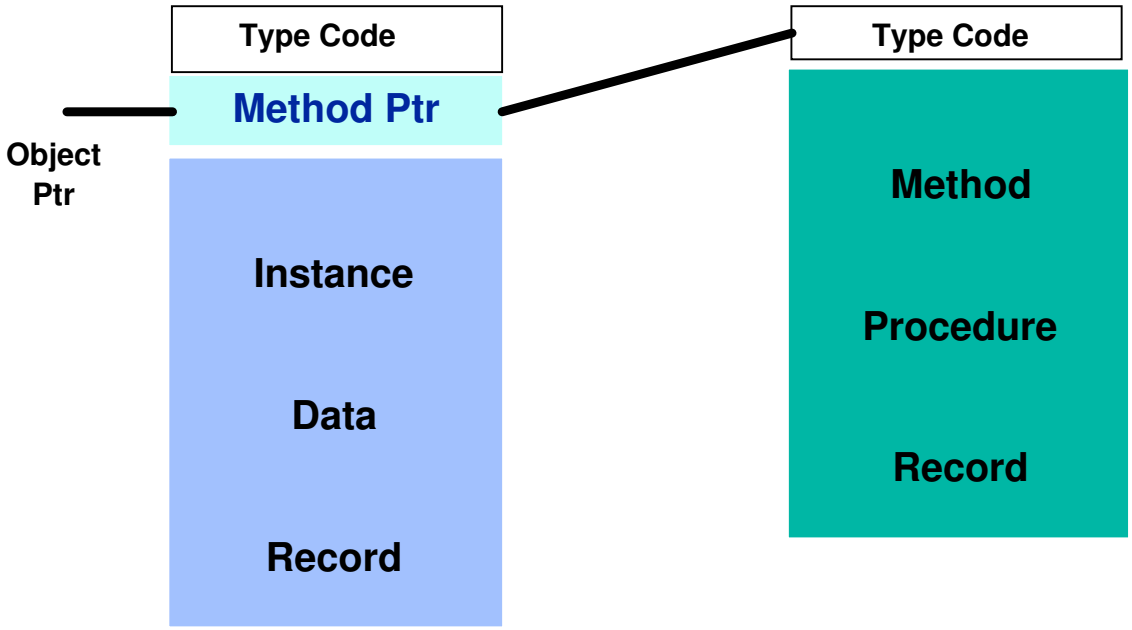

Figure 2. Modula-3 Object Model

The Modula-3 object model (Figure 2) is slightly simpler than the ATK object model because the compiler and runtime work together to simplify the details.

The target of an object pointer is a single record. The first field of the record is a pointer, called the *method pointer*. The remaining fields are the data fields of the object in the order of their declaration. The target of the method pointer is a copy of the type code for this object type[5]. This type code is immediately followed by a record of typed procedures. These procedures are the methods of the object and occur in the order in which they are declared.

In the Modula-3 environment, ATK objects appear as untraced objects because the C implementation uses, in effect, untraced references and objects. At a later stage of this work, one could modify the C implementation to use the Modula-3 storage management system to make the objects and references traced.

3. Runtime Environments

The ATK runtime consists of about 2500 lines of strongly target dependent C code that has a number of very fragile passages. A number of functions provided by this code must be replicated in the Modula-3 environment so that the ATK C implementation will work correctly.

---

[5] This second copy of the type code is a historical accident and is likely to disappear in later implementations Modula-3 including later versions of the SRC Modula-3 system.

There is a procedure that takes a single C string parameter and returns an instance of the class named in the string. The code for the class will be dynamically loaded before instantiation if it is not yet part of the memory image of the running program.

The ATK runtime contains a procedure to compare the types of two objects. This procedure is implemented by comparing the class info pointers of the two objects. This is equivalent to comparing the type codes of the two objects in Modula-3.

There is a procedure to convert a class info pointer into a type name (as a C string) and a procedure that returns the name of the type of an object as a C string.

This section describes the design of less than 300 lines of safe Modula-3 code that implement the ATK runtime environment except for dynamic loading. This code uses only the trivially portable part of the src Modula-3 system.

3.1 Type cells

The Modula-3 compiler constructs a type cell for each traced reference type and each traced or untraced object type that is compiled. This type cell contains all of the information needed to instantiate or destroy traced instances of the type. Untraced objects that contain dynamically allocated untraced storage require some user written code.

Modula-3 execution initialization creates an array of references to type cells. The index of the pointer to the type cell of a type is the type code of the type. Here is the part of the declaration of a type cell that is of interest in this work:

```
TYPE Typecell = RECORD
  typecode: Typecode;   (* a CARDINAL *)
  lastSubTypeTC: Typecode;
  . . .
  traced: BOOLEAN;
  . . .
  defaultMethods: ADDRESS;
  . . .
  initProc: TypeInitProc;
  . . .
  brand: String;   (* A char* in C. *)
  name: String;
  . . .
END;
```

The type code of any type is less than the type code of all subtypes of the type. Moreover, the values of the fields typecode and lastSubTypeTC of a type cell are such that all of the subtypes of the type have type codes between typecode and lastSubTypeTC (inclusive). In addition, only subtypes of the super type have type codes in this range. This can be stated somewhat more precisely as follows:

Let `T` and `U` be types, let `T_tc` be the type cell of `T` and (in Modula-3 syntax), let the type code of `U` be `TYPECODE(U)`. Then,

`U <: T` ***iff*** `T_tc.typecode <= TYPECODE(U) <= T_tc.lastSubTypeTC.`[6]

---

[6] In Modula-3, the symbol <: is used to name the relationship *is a subtype of.*

The defaultMethods field of a type cell is non-NIL only for object types. In this case, it is a pointer to a record of method procedures. The methods appear in the order in which they are declared, from the root of the relevant class hierarchy to the object type whose type cell contains the pointer.

The definition of Modula-3 requires that all data values be initialized to some legal value for the type. In addition, initial value specifications are permitted in type declarations and program authors are encouraged to use this part of the language to improve the reliability of programs. After an instance of a type is allocated, the procedure initProc is called to initialize the data of the type instance.

The name field of a type cell contains the fully qualified name of the type. For example, the type T declared in interface View would have the C string `View.T` in this field.

3.2 Modula-3 Type ATKObject

There are basically two reasons for introducing a Modula-3 type that is the super type of all ATK objects in the Modula-3 environment. Introducing this type makes it very easy to determine if an arbitrary Modula-3 object is an ATK object or some other kind of object. The second is somewhat more complicated: it provides a mechanism for bringing the essential class procedures of the ATK objet model into the Modula-3 environment.

ATK is implemented under the assumption that all classes have four class procedures, New, InitializeObject, FinalizeObject, and Destroy, even if the procedures are empty. These procedures are essential for the correct creation and destruction of objects.

The class procedure InitializeObject initializes object data that is unique to this class (and which does not occur in any super class) and then calls the InitializeObject procedure of its immediate super class. The procedure New allocates storage for the object specific part of an atk object, fills in the pointers described above and then calls the InitializeObject procedure of this class.

The procedure FinalizeObject deallocates storage that is specific to the current class (and not to its super classes) and then calls the FinalizeObject procedure of its super class. The procedure Destroy calls FinalizeObject and then deletes the allocated storage for the instance specific data.

The root of the tree of ATK objects in the Modula-3 environment is an untraced object type with four methods:

```
INTERFACE ATKObject;
CONST
  True: Boolean = 1;
  False: Boolean =0;
  BadAddress: ADDRESS = NIL;

TYPE
  Boolean: INTEGER;
  file_star = ADDRESS;

EXCEPTION Error(Methods); Fatal(Methods);

TYPE T = UNTRACED OBJECT
  METHODS
    new(): T RAISES {Error} := New;
    init() RAISES {Error} := Init;
    destroy(): T RAISES {Error} := Destroy;
    getReadyToDie() RAISES {Error} := GetReadyToDie
  END;

  (* Declarations of Method Procedure Signatures
     and type Methods have been elided. *)

END ATKObject.
```

All Modula-3 ATK object types are subtypes of `ATKObject.T`. Note that this means that the first four methods of all atk objects are new, init, destroy and getReadyToDie. The method new provides the function of the class procedure New, the method init the function of InitializeObject, the method destroy the function of Destroy and the method getReadyToDie the function of FinalizeObject[7]. At each level of the class hierarchy, these four methods are overwritten in a purely mechanical fashion.

In addition to providing an object type with these four methods, the interface ATKObject provides a few types that are used throughout the entire Modula-3 translation of ATK object type declarations. The type Boolean serves to provide C booleans in Modula-3, that is, 32-bit data that may have value 0 or 1. The Modula-3 type BOOLEAN is an enumerated type with two elements and, therefore, occupies one byte. A number of atk object methods accept parameters that are of C type FILE*. The type file_star is used to represent this type in Modula-3.

3.3 Modula-3 ATK Runtime

Sections 3.1 and 3.2 have described all of the pieces needed to construct a Modula-3 version of the ATK runtime. The resulting small and safe definition depends on only one Modula-3 implementation detail in one line of implementation code. Here is the declaration of the Modula-3 ATK runtime.

---

[7]The first named author finds the pseudo-English word finalize to be a most distressing abuse of the English language because it is used to suggest that a piece of work is almost completed when this is usually far from true. Moreover, even if this abuse of English were correct, it would be inappropriate in this case because the object is being prepared for destruction not completed in any sense.

```
INTERFACE ATKRuntime;
  IMPORT ATKObject, RT0;
  EXCEPTION Error(Code); Fatal(Code);

PROCEDURE NewFromTypeCode(
  tc: RT0.Typecode): ATKObject.T RAISES {Error};

PROCEDURE NewFromName(
  name: RT0.String): ATKObject.T RAISES {Error};

PROCEDURE TypecodeFromName(
  m3name: TEXT): RT0.Typecode RAISES {Error};

END ATKRuntime.
```

Given the type code, tc, of an ATK object that is to be instantiated, execution proceeds as follows:

(1) The typecode, tc, is used to obtain the correct type cell. This is a safe operation.

(2) The defaultMethods field of the type cell together with the knowledge that the first method of the object type is the New procedure is used to call New to instanticiate the object. This one step (corresponding to one line of source code) depends on the details of the SRC Modula-3 implementation.

(3) The newly created and initialized object instance is returned.

Object instance destruction does not require this indirect activity because it can be done by invoking methods of the objects.

The procedure NewFromName calls procedure TypecodeFromName to determine the type code of the type named by its actual parameter. The value returned by this call is the actual parameter of NewFromTypeCode and the return value of this procedure is returned.

The procedure TypecodeFromName performs table lookup on the type cell array to find a type cell whose name field matches the actual parameter. When such a type cell is found, the index of it's pointer in the type cell array is returned; this is the type code of the type with this name.

When Modula-3 code is emitted by class, the `.ih` and `.eh` files contain macros for class procedure invocation that actually call the procedure attributes of the ATKRuntime interface. The same applies to the macros that are used to invoke the class procedures Destroy and FinalizeObject.

4. Modula-3 Translation of ATK Class Declarations

Mechanically translating the ATK class declaration files (`.ch` files) into Modula-3 raised many technical questions related to the original implementation of ATK which are described below. Here, it is assumed that these problems have been solved. An outline of the textual layout of a `.ch` file is shown in Figure 3. The initial part of the file contains an arbitrary number of C typedefs, C declarations and C preprocessor directives such as #define and #include (not to mention arbitrary C declarations and #ifdef preprocessor directives). This is followed by a class declaration that includes the name of the class being declared and the super class of this class. Anonymous types may be declared almost anywhere in the class declaration. The class declaration may be followed by additional C type declarations, data declarations or preprocessor directives.

```
struct foo {
  union { ...} whatever;
  long (*foo)();
};
```

```
class whatever: super {
  ...
}
```

```
#define BAR 0x77
#define WHAT (THIS | THAT)
```

Figure 3. Structure of Class Declaration (.ch) File.

At this level of the ATK implementation, the type information in places like struct declarations and method signature declarations are optional comments. The traditional class preprocessor ignores such information.

Rather than giving a set of formal translation rules, this section illustrates the transations that are performed by means of a simple example. Enhancements to the class preprocessor that were needed to perform this translation are described in Section 5, below.

4.1 A Simple ATK Class Declaration

Here is a declaration of a truly simple ATK class that serves as a way of describing most of the translation problems that are encountered assuming that the messy details mentioned above have been dealt with.

```
class atom {
classprocedures:
  Intern(char * name) returns struct atom *;
  InitializeClass() returns boolean;
  InitializeObject(struct atom * self) returns boolean;
  FinalizeObject(struct atom *self);
macromethods:
  Name() returns char* = (self-> name);
data:
  char *name;
}
```

This (modified) declaration of atk root class atom shows many of the properties of the class declaration language. The source file declares a class atom which would appear in Modula-3 as object type AAtom.T. This class has four class procedures, InitializeObject and FinalizeObject which were described above, a class initialization procedure InitializeClass, and a class procedure Intern which takes a char* parameter and returns an instance of the class.

Class initialization procedures are called at most once: before any instances of the class are created. They perform whatever global initialization is required to support the creation and deletion of class instances. In the case of class atom (a rather specialized situation), class initialization creates a hash table that is used to contain the strings that occur in instances of atom. The hash table is used to make sure that there is at most one atom whose name field contains a particular string.

When this .ch file is processed for Modula-3 by the new class, a total of five output files are produced. A .ih file is an include file for C clients of atom.ch. This file differs from the traditional C file in that the emitted macros refer to the Modula-3 runtime environment and use the Modula-3 procedure and data names. A similar statement applies to the emitted .eh file which is included by the C implementors of the class.

The C include files are quite straightforward given the Modula-3 procedure and data naming conventions and are not discussed further here. The remainder of this section contains descriptions of the emitted Modula-3 files.

4.2 Modula-3 Public Interface

```
UNSAFE INTERFACE AAtom;
IMPORT ATKObject, Ctypes;

TYPE
  T_public = ATKObject.T OBJECT
    METHODS
      name(): Ctypes.char_star
    END;

  T <: T_public;

<* EXTERNAL "atom__Intern" *>
PROCEDURE Intern(
  junk := ATKObject.BadAddress;
  name: Ctypes.char_star): AATom.T;

END AAtom.
```

There are a number of transformations that were performed in constructing the Modula-3 public interface. Some of these transformations were motivated by a desire to provide unique file names for files containing declarations of ATK object types and others were motivated by a desire to have the emitted Modula-3 code closely comply with widely used spelling conventions for Modula-3.

The first transformation on the ATK class name is to convert the first letter of the class name to upper case to comply with the convention that the first letter of the name of an interface is an upper case letter. Further, an initial upper-case A was added to the name to make sure that all ATK interfaces and implementations have names that are different from other names in the Modula-3 library.

The interface follows the conventions that call for method names to begin will lower-case letters and for the principal type of the interface to be named T.

There are many extremely unsafe constructs in the data part of ATK objects. It appears to be desirable to limit the scope of these unsafe constructs to existing code and, therefore, the data attributes of objects do not appear in the public interface.

Macros are a rather unsafe construct that is peculiar to C and to some older languages at the same level; they do not appear in Modula-3. Therefore, each macro method must be converted to an actual procedure that can be bound to a Modula-3 method. In this spirit, the macro method name was converted to an actual procedure which is bound to the method name. This transformation is done by the new version of class.

The attributes of an atom that are visible to clients of AAtom are the attributes of T_public. The last declaration indicates that an AAtom.T is an opaque subtype of T_public which must be revealed somewhere in any runtime unit that contains clients of AAtom.

The class procedure Intern appears as a procedure attribute of the public interface. The procedure appears as an external declaration that binds the existing C code for function atom__Intern to the Modula-3 procedure AAtom.Intern. The first parameter of the C function is irrelevant and, therefore, the parameter junk with a default initial value of NIL is used to match up the calling sequences of the Modula-3 code and the C code.

If class atom had any actual methods these methods would also appear in the methods section of the declaration of type T_public.

The Modula-3 translation of classes further from the root will have additional methods and overrides in their public and private declarations.

4.2 Modula-3 Private Interface

```
UNSAFE INTERFACE AAtomP;
IMPORT AAtom, ATKObject, Ctypes;

REVEAL AAtom.T = AAtom.T_PUBLIC BRANDED OBJECT
    Name: Ctypes.char_star;
  OVERRIDES
    name := Name;
    new := New;
    init := Init;
    getReadyToDie := GetReadyToDie
END;

<* EXTERNAL "atom__FinalizeObject" *>
PROCEDURE FinalizeObject(
  junk := ATKObject.BadAddress; self := AAtom.T);

<* EXTERNAL "atom__Name" *>
PROCEDURE Name(self: AAtom.T): Ctypes.char_star;

PROCEDURE Init(self: AAtom.T) RAISES {ATKObject.Error};
PROCEDURE New(
  self: AAtom.T): ATKObject.T RAISES {ATKObject.Error);
PROCEDURE GetReadyToDie(
  self: AAtom.T) RAISES {ATKObject.Error};

(* The following appears in the private interface if
   there is a InitializeClass procedure in the ATK
   class declaration. *)

<* EXTERNAL atom__InitializeClass *>
PROCEDURE InitializeClass(): ATKObject.Boolean;

END AAtomP.
```

The private interface for ATK class atom, AAtomP, contains a revelation of the public type T as a subtype of type AAtom.T_public. There is a data field Name which contains the string associated with the atom; this is the data field in the .ch file. There are overrides for four methods: The method name with an unspecified default from the public interface. This method is bound to the external C function atom__Name which is the C function that is equivalent to the macro method name declared in the .ch file.

The methods new, init and getReadyToDie, inherited from the parent type, ATKObject.T are overridden by specific functions. The signatures of the functions are declared in this interface and the implementations are provided in the implementation module.

An external declaration for the C class procedure atom__FinalizeObject appears in this interface so that it can be use in the implementation module.

4.3 Modula-3 Implementation Module

Almost all of the implementation of an ATK class is the original C code. However, a few details such the implementation of the methods new, init and getReadyToDie are in mechanically generated Modula-3. This section describes this generated Modula-3.

```
UNSAFE MODULE AAtom EXPORTS AATom, AAtomP;
IMPORT ATKObject;

PROCEDURE Init(self: T) RAISES {ATKObject.Error} =
BEGIN
  IF InitializeObject(self := self) =
       ATKObject.FALSE THEN
    RAISE ATKObject.Error(ATKObject.Methods.Init) END;
  ATKObject.T.init(self)
    (* init method of parent class *)
END Init;

PROCEDURE GetReadyToDie(
  self: T) RAISES {ATKObject.Error} =
BEGIN
  FinalizeObject(self := self);
  ATKObject.T.getReadyToDie(self)
    (* getReadyToDie method of parent class. *)
END GetReadyToDie;

PROCEDURE New(self: T): T RAISES {ATKObject.Error} =
BEGIN
  RETURN NARROW(ATKObject.T.new(self), T)
END New;

BEGIN
  (* The following appears in implementation modules when
     there is a InitializeClass procedure in the ATK
     class declaration. *)
  IF InitializeClass() = ATKObject.False THEN
    RAISE Fatal(Methods.ClassInitialization) END;

END AAtom.
```

The generated Init procedure calls the InitializeObject class procedure of this class declaration, if there is one, and then calls the init method of the super class. This follows the ATK specification for object initialization.

The generated GetReadyToDie procedure calls the FinalizeObject class procedure of the ATK class and then calls the getReadyToDie method of the immediate super type in accord with the specification.

The New procedure simply calls the new method of ATKObject.T passing an instance of the current type as a parameter. The actual method will create another instance of this type and return it. This return value is the return value of the procedure.

If there is a class initialization procedure in the .ch file, a call to this procedure is added to the module initialization block so that the call is executed before any attribute of any instance of the type is created or referenced.

The remainder of this manuscript is not available at this time.

A complete version will be submitted for publication at a later date.

5. Enhancements to the ATK Class Preprocessor

5.1 Type Declarations

Top Level Only

No Anonymous Types

5.2 Explicit Imports Required

import AAtom;

5.3 Preprocessing is Order Dependent

Introduced Special Qualifier to Break Circular Dependencies

5.4 Opaque Type Modifier

Type is "fully declared" only in C; revealed as UNTRACED BRANDED REF INTEGER

5.5 Private Type Modifier

Type is declared only in C

5.6 Rewrite/Transformation Rules

String substitution for Modula-3 Generation

Modula-3 Type Naming

5.7 Procedure Signature Must be Declared

5.8 Method, Macro Method and Macro Signatures are Required

5.9 No anonymous types.

6. Modifications to Existing Class Declaration Files

6.1 Source Contains Excellent Examples of How to Write Bad C

Might be Boolean
- bit field
- (unsigned) int
- (unsigned) char
- (unsigned) short

Might be void*
- long
- char*
int *

Return types:
long == void* ==
char* == int == void

Anything resembling integer may be "Word"

7. Lessons and Conclusions

7.1 Class and C Language Issues

A language that makes it easy to write bad programs make it easy to write bad systems

Parameter type checking omitted.

Implicit Opaque type declarations (struct *)

No validation of compile or runtime revelation

Arbitrary compilation order inhibts inter-compiliation unit checking

Textual Substitution Macros make source unreadable

Unions to optimize use of PDP-11 address Space.

7.2 Lessons We Learned

Original ATK objet model was well done.

Things are cleaner if you have objects with a relatively small number of methods. (200+ in a base class is far too many)

Checking types helps find lots of bugs

Machine generated macros can be used to hide a multiltude of sins

Similar techniques could be used to transform single parent inheritance C++ into Modula-3

7.3 Things We Would Do Differently

Trash the old class preprocessor code

Design a real parser

Write the program in Modula-3

Start *much* earlier!

7.4 Conclusions

We can change object models without changing the application code.

Type information at runtime is essential.

C is a bad language because it lets silly mistakes accumulate for years until your software becomes unchangeable. (C++ is worse)

Use of procedural macros enabled object model transformation.

Textual macros are evil!