Combining Language and Database Advances in an Object-Oriented Development Environment

Timothy Andrews & Craig Harris

Ontologic, Inc.
47 Manning Road
Billerica, MA 01821

"You're both right. It's a dessert topping and a floor wax!"
– Chevy Chase

Abstract

Object-oriented languages generally lack support for persistent objects—that is objects that survive the process or programming session. On the other hand, database systems lack the expressiblity of object-oriented languages. Both persistence and expressibility are necessary for production application development.

This paper presents a brief overview of VBASE, an object-oriented development environment that combines a procedural object language and persistent objects into one integrated system. Language aspects of VBASE include strong datatyping, a block structured schema definition language, and parameterization, or the ability to type members of aggregate objects. Database aspects include system support for one-to-one, one-to-many, and many-to-many relationships between objects, an inverse mechanism, user control of object clustering in storage for space and retrieval efficiency, and support for trigger methods.

Unique aspects of the system are its mechanisms for custom implementations of storage allocation and access methods of properties and types, and free operations, that is operations that are not dispatched according to any defined type.

During the last several years, both languages and database systems have begun to incorporate object features. There are now many object-oriented programming languages. [Gol1983, Tes1985, Mey1987, Cox1986, Str1986]. Object-oriented database management systems are not as prevalent yet, and sometimes tend to use different terms (Entity-Relationship, Semantic Data Model), but they are beginning to appear on the horizon [Cat1983, Cop1984, Ston1986, Mylo1980]. However, we are not aware of any system which combines both language and database features in a single object-oriented development platform. This is essential since a system must provide both complex data management and advanced programming language features if it is to be used to develop significant production software systems. Providing only one or the other is somewhat akin to providing half a bridge: it might be made structurally sound, perhaps, but it is not particularly useful to one interested in getting across the river safely.

Object-oriented languages have been available for many years. The productivity increases achievable through the use of such languages are well recognized. However, few serious applications have been developed using them. One reason has been performance, though this drawback is being eliminated through the development of compiled object languages. The remaining major negative factor, in our view, is the lack of support for persistence; the lack of objects that survive the processing session and provide object sharing among multiple users of an application.

Database management systems, in contrast, suffer from precisely the opposite problem. While having excellent facilities for managing large amounts of data stored on mass media, they generally support only limited expression capabilities, and no structuring facilities.

Both language and database systems usually solve this problem by providing bridges between the systems. Thus the proliferation of 'embedded languages', allowing language systems to access database managers. These bridges are usually awkward, and still provide only restricted functionality. Both performance and safety can be enhanced through a tighter coupling between the data management and programming language facilities.

It is this lack of a truly integrated system which provided our inspiration at Ontologic, Inc. This paper reviews Ontologic's VBASE Integrated Object System and describes how it combines language and database functionality.

1. General System Overview

The single overriding consideration which drove the design and development of VBASE was to provide a complete development system for practical production applications based on object-oriented technology.

Two goals flowing from this motivation were:

1) To integrate a procedural language with support for persistent objects. This support should be as transparent as possible to users of the system.
2) To take maximum advantage of strong typing inherent in object systems in both the language and database.

The system derives its heritage from many precursors. Probably the single most important language influence was the CLU programming language developed at MIT[Lis1981]. Thus, VBASE is based around the abstract data type paradigm, rather than the object/message paradigm. This orientation manifests itself in many areas. For example, in typical object/message systems, all access to object behavior is through a uniform message syntax. In VBASE, object behavior is elicited by a combination of properties and operations. Properties represent static behavior; operations represent dynamic behavior. Property definition and access are syntactically differentiated from those of operations. This provides a more natural model of object behavior. It also saves the programmer from writing trivial code to get and set the values of properties. In VBASE, these operations are normally generated by the system, further increasing programmer productivity.

In fact, the fundamental emphasis on a strong separation between the specification of a system and the implementation of a system is common to abstract data type and object/message systems. VBASE uses this methodology at several levels to provide an extremely flexible architecture. (The overall architecture is shown in figure 1.)

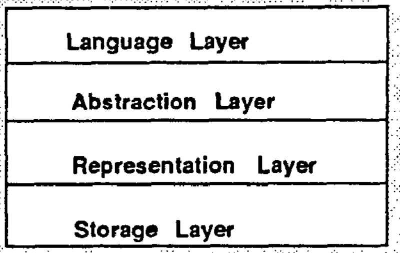

Figure 1. VBASE Architecture

The language layer contains compilers for defining and implementing the behavior of objects. The abstraction layer implements the object meta-model, providing support for inheritence, operation dispatching, method combination, and property manipulation. The representation layer is the locus of our reference semantics. The storage layer is responsible for object persistence.

Each layer of VBASE is implemented within VBASE itself. Thus each layer has a VBASE specification and a VBASE implementation. Consequently, we realize many of the advantages of the system in the implementation of the system itself. Chief among these has been the ability to implement the total system quickly and then tune performance by replacing or enhancing various implementations. As the specifications were unaffected, effort could be concentrated where it was needed, allowing a good deal of performance work to be completed on the system at an early date.

As mentioned previously, support for persistent objects, or objects that survive process lifetimes and programming sessions was a key motivation. This requires 'database' support; that is, handling of storage on stable media such as disks. There are some further capabilities implied by 'database' support:

1) sharing of object data among multiple processes/users.
2) handling large numbers of objects and consequently handling a large object storage space.
3) software stability so that the object space is maintained in a consistent state in the face of system or media failure.

We had a further desire: to provide 'seamless' support for persistent objects through a natural syntax. We particularly wished to avoid the 'embedded language' approach. The goal was to integrate persistent objects completely into the language as pseudo-standard variables. This makes the entire expression processing capability of the language layer available to the persistent objects.

The last of the primary motivations directing the design and development of VBASE was to build an object system that also provided strong typing. One of the most serious drawbacks of the Smalltalk class of object systems is their lack of any notion of type specificity. There are simply objects. The benefits of strong typing are well known. There are three that are especially important in a system intended for commercial development.

First, strong typing resolves many more errors at compile time than weakly-typed systems. Since objects' main claim to fame is productivity gain, the resolution of errors at an earlier time in the software lifecycle is significant. It should also be noted that compile time errors are generally easier to analyze and correct than errors of type mismatch that occur at run time.

Second, strong typing of object data structures provides superior specification of the system. Rather that relying on user-constructed naming conventions to convey type information (aFruit, anApple, aCar, etc), data structure type declarations provide a clear and exploitable specification since the type declarations are all part of the system specification. Thus, one can examine the declaration aFruit: Fruit, and then examine the Fruit definition for further information. This process can be applied recursively to any desired level of detail, and is not dependent on adoption of any conventions by the system implementors.

One final issue regarding strong typing is its effect on system performance. A strongly typed system allows the language processor to do far more analysis at compile time. This analysis can often reduce the need for runtime type checks, as well as allow methods to be statically bound. Our experience to date indicates that 90% of all type checking and method binding can be done at compile time. This eliminates the performance degradation frequently resulting from object systems' need to dynamically bind all method code to achieve object/message behavior. Thus VBASE exhibits the functionality of dynamic method binding based on a hierarchy of types, as do all object systems. However, it does so with performance comparable to a compiled, statically bound system.

There are, in fact, many more optimizations that the language processor can do based upon its knowledge of the semantics of the types and objects in the system, and the amount of work that can be done is proportional to the amount of information that is contained in the definitions. This is yet another argument for a strongly typed system.

While VBASE supports strong typing, the VBASE type system also provides a great deal of flexibility.

Through the use of subtyping and parameterized types and operations, VBASE provides a high degree of static type checking. There are cases, however, where static type checking is impossible or undesirable. Explicit run-time type checking can then be used to achieve the same expressive capability as an untyped object system. Thus, in VBASE, the tradeoff between compile-time optimization and run-time flexibility is controlled by the application developer.

2. System Components

VBASE is currently implemented on top of Sun OS 3.2 UNIX. There are presently two language interfaces: TDL (for Type Definition Language) and COP (for C Object Processor). TDL is used to specify a data model. That is, it is used to define data types and specify their associated properties and operations. COP is used in two roles. It is used to write the code to implement the operations. It is used to write the applications programs. There is also a set of tools to assist development. These include a debugger, interactive object editor, and a verifier program that checks consistency of the physical layout of the object data space.

TDL is a proprietary language. It is block structured, with features in common with such languages as Pascal, Modula, and Algol. Types are the most common entities defined in TDL. A type serves as the nexus for behavior of its instances. It determines the properties for which its instances supply values and it defines operations which may be performed on its instances.

The current version of the system allows only one supertype to be specified; this supertype places the type definition in the type hierarchy. Behavior is inherited via the type hierarchy in the expected manner. A type is also a block scope in TDL, and consequently may contain other arbitrary definitions along with its central property and operation definitions.

COP is a strict superset of the C language as defined by Kernighan and Ritchie. Any program which compiles with standard C will compile with COP. It contains syntactic extensions to the C language to allow typed declarations of variables, access to properties of objects, and invocation of operations on objects. COP is used to write the actual code that implements the operations and properties and other behaviors specified in TDL. It is currently implemented as a preprocessor which emits standard

C code.

The debugger allows source line debugging of the COP source code. The object editor allows interactive traversal of type and object definitions, assignment of object property values, and invocation of operations. The storage verifier examines the physical layout of the database and verifies its structural integrity.

3. Interesting Language Aspects

VBASE incorporates most of the standard object technology. There is a taxonomy of types, with subtypes inheriting both properties and operations from their supertype. Subtypes can add more specific behavior by specifying additional properties or operations, and can also refine existing behavior of inherited properties or operations. When an operation is invoked, it is dispatched according to the type of the object of the invocation. Thus in COP one writes:

```
Entity$print (someObject);
```

which means find the print operation of the type closest in the type hierarchy to the direct type of someObject. Of course, there is complete type extensibility, with the user being able to define and use whatever types are desired.

3.1 Strong Typing

The most important language influence—strong typing—is unusual among current object systems. In TDL, the following definition illustrates the typing aspects:

```
define Type Part
  supertypes = {Entity};

properties = {
    partID: Identifier;
    name: optional String;
    components: distributed Set[Part]
      inverse componentOf;
    componentOf: Part inverse components;
}
operations = {
      display (p:Part)
        raises (NoDisplayImage)
        method (Part_Display);

      isComponentOf (p1: Part, p2:Part)
        raises (IsRootComponent)
        method (Part_isComponentOf)
        returns (Boolean);

      connect (p: Part, to: Part,
        keywords
          optional using: Connector)
        raises (BadConnect)
        method (Part_Connect)
        returns (Part);

      iterator components (p: Part)
        yields (p: Part)
        method (Part_Components);

      refines delete (p: Part)
        triggers (Part_deleteTrigger);
      };

    PRIVATE
    properties = {
      displayImage: optional Image;
      isRootComponent: Boolean := False;
      };

  end Part;
```

```
    define Type Pipe
      supertypes = {Part};

      properties = {
        length: Integer := 0;
        diameter: Integer := 0;
        leftConnection: Part;
        rightConnection: Part;
        threadtype: optional ThreadType :=
            Thread ype$ScrewThread;
        isInsulated: optional Boolean:= False;
    };
      operations = {
        refines connect (p: Pipe, to: Part,
          keywords
            optional usingConnector: Connector)
          raises (ThreadtypeMisMatch,
              IncompatibleMaterials)
          method (Pipe_Connect)
          returns (Part);

        materialsCompatible(p1:Pipe, p2:Pipe)
          method (Pipe_MaterialsCompatible)
          returns (Boolean);

        refines delete (p: Pipe)
          raises (CannotDelete)
          triggers (Pipe_deleteTrigger);
        };
    end Pipe;

    define Type ThreadType is enum (ScrewThread,
PolThread);
```

Figure 2. TDL code for two type definitions is shown. Type Part is a generic definition used for all parts in this hypothetical engineering design database. It is the supertype of type Pipe which inherits all the behavior of parts and adds behavior specific to pipes.

Note that all definitions are associated with a type. This applies at all levels. At the topmost level, the 'define Type Part' fragment says that Part is of type Type. In a similar manner, properties are defined in terms of their data type, as are operation arguments, return values, and exception specifications. The type information contained in the TDL specifications is then used to generate a schema for an object database. In practice, VBASE provides a substantial kernel schema. TDL is then used to augment this schema with user extensions. Thus TDL can be described as an incremental schema compiler.

After the datatypes are defined, COP code is written and compiled against the object database. The COP compiler is a database application, and uses the type information of the database to do what type checking is possible at compile time. When static type checking is not possible, the check is deferred to runtime.

Figure 3. COP code for the Pipe_Connect method. This method implements the operation defined in Type Pipe in figure 2.

```
method
obj Part
Pipe_Connect (aPipe, toPart, usingConnector)

obj Pipe aPipe;
obj Part toPart;
keyword obj Connector usingConnector;

{
   obj Part connectedPart;
   obj Pipe toPipe;
   int j;

   if (hasvalue (usingConnector))
   {
      connectedPart = Pipe$Connect
         (aPipe, usingConnector);
      return (Part$Connect (connectedPart, toPart,
using: usingConnector));
   }

   toPipe = assert (toPart, obj Pipe);
   except (ia: IllegalAssert)
   {
      PipeSystem$ErrorPrint (aPipe, toPart, "can only
connect Pipes to Other Pipes");
   }
   if (aPipe.threadtype != toPipe.threadtype)
      raise (ThreadTypeMisMatch);

   if (! Pipe$MaterialsCompatible (aPipe, toPipe))
      raise (IncompatibleMaterials);

   aPipe.leftConnection = toPipe;
   toPipe.rightConnection = aPipe;

   connectedPart = $$ (aPipe, aPart);

   return (connectedPart);
}
```

Operations in the type definition are implemented by methods written in COP. Figure 3 shows the code to implement the Pipe_Connect method.

Note that all object variables are declared with the additional keyword obj. This allows the program variables to be associated with types in the schema, and COP can then do type checking based on the schema information. Therefore the assignment:

```
aPipe.leftConnection = toPipe;
```

is allowed since the leftConnection property of type Pipe is of type Part, and the declared type of the variable toPipe is Pipe. These are compatible since Pipe is a subtype of Part. All operation invocations and their arguments are similarly checked.

When it is not possible to determine type compatibility at compile time, the programmer uses the assert statement. The assert statement defers type checking until runtime. This allows handling assignment of a more general type to a more specific type without violating strict compile-time type checking in most cases, an invaluable productivity win in large, complex systems. In the method example in figure 3, assert is used for two purposes.

First, this simple implementation assumes that pipes can only be connected to pipes; thus the assert does a runtime type check for the programmer, while allowing the code to be written for the more general case of connecting a pipe to any other part.

Second, the statement:

```
toPipe.rightConnection = aPipe;
```

will not compile if the declared type of toPipe is Part. This is because Part does not define the property rightConnection. rightConnection is defined by Pipe. Thus while it is quite possible that, since part is pipe's supertype, a given part is a pipe, it is not guaranteed by the declarations of the program. This is a very common type violation in object systems, and this is what VBASE prevents. The assert alerts the implementor that the actual type of the object toPipe must be pipe (or a subtype of pipe) in order for this assignment to be valid.

This is in stark contrast to Smalltalk-like object systems.

3.2 A Block-structured Schema Definition Language

Another notable feature of TDL is that it is a block structured language. This is different from most object systems, and certainly very different from most schema definition languages of DBMS's. It gives TDL the kind of complex name environment that most structured programming languages have, with the concommitant reduction of name conflicts. It also means that the system supports pathnames, allowing simple grouping of names, and relative names as well as global names.

The '$' is the pathname component separator. Thus the name 'Threadtype$Screwtype' refers to the name Screwtype within the block defined by the name Threadtype. Analogously, in the COP fragment, a name such as 'Pipe$Connect' refers to the name Connect within the Pipe scope. Names which begin with a '$' are considered absolute relative to the root scope maintained by VBASE. Names in the root include such things as kernel type names, system exception types, etc. New type definitions, unless contained within a scope, are placed in the root. A file is not considered a scope. This last point is generally not true of present systems, and has the pleasant side effect that many definitions can be placed in a single file or set of files without affecting the desired scoping.

3.3 Constant & Variable Definitions in TDL

Another programming language capability available in TDL is defining constants and variables. This allows user-customized constants to be placed in the object database. For instance:

```
define Constant myDefault := Null;
define Constant No := False;
define Constant Yes := True;
define Variable background: Color := Colors$Gray;
```

As in other languages, constants are immutable, while variables can have their bindings reassigned.

3.4 Enumeration, Union, and Variant Types

Along similar lines, TDL allows the definition of enumerations, unions and variants. Type definitions such as these are rarely supported in object systems or database systems. This is certainly unfortunate, as their uses are well known. For instance:

```
define type Day is enum (Monday, Tuesday, ...);
```

Unions and variants are especially important in object systems, particulary those which do not support multiple inheritance. These definitional abilities allow any type of polymorphism desired to be expressed without circumventing the type safety of the system. Rather than having to declare any variable that can potentially hold objects of types which are disjoint in the type hierarchy as Entity, one can restrict the set of types to only those which can actually occur:

```
define type BlockScope is union (Type, Module,
Environment, Directory, ...);
```

This allows common mistakes to be ferreted out at compile time. Thus:

```
obj BlockScope scope;
....
scope = Day$Monday;
```

will fail at compile time. If, instead, the variable's scope had to be declared of type Entity (the root of the type hierarchy), this mistake would go unnoticed during compilation.

3.5 Parameterization

Another significant capability of VBASE is what is sometimes referred to as parameterization: the ability to specify the type of the objects contained inside aggregate objects. This ability is often not even available in procedural languages. Thus one can write:

```
obj Array[Animal] myZoo;
```

VBASE will type check all insertions into and assignments from the aggregate just as it checks standard types. The lack of such checking is a serious shortcoming of present systems as aggregates are widely used to store critical system information. Often this information is typed, but there is no way to enforce proper use short of writing expensive runtime checks of the elements of the aggregate. Thus:

```
{
  obj Array[Animal] myZoo;
  obj Fruit aKiwi;

  aKiwi = myZoo[3];
}
```

fails at compile time.

Once again, it must be emphasized that this does not limit the programmer. If the programmer actually wishes an 'untyped' Array:

```
obj Array[Entity] myUntypedArray;
```

will suffice. However, as most system implementors can testify, this is rarely the case. More usually,the appropriate types cannot be defined within the confines of the chosen system, and the programmer must circumvent the system in order to accomplish the task with reasonable efficiency.

3.6 Method Combination

The above paragraphs have described some of the more interesting 'data definition' (to use the term rather loosely) capabilities of VBASE. There are also some very interesting runtime features in VBASE. Perhaps the most notable of these is the VBASE approach to method combination. Method combination in object systems results when a refining method invokes its refinee. In Smalltalk, for example, one uses the pseudo-variable 'super' for this purpose. VBASE uses '$$' for this purpose. This notation, rather than super or some derivative thereof, was chosen because of the novel view of operations behavior that VBASE takes.

Operations are viewed as being implemented by a series of executable code fragments. The number of fragments is arbitrary, and is the sum of all triggers and methods defined in the operation. Reviewing the TDL figure 2, note that each operation definition can include a method clause, and a triggers clause. Each operation is therefore potentially associated with one method, called the base method, and an arbitrary number of trigger methods. The execution sequence begins with the first trigger in the triggers clause. The '$$' syntax transfers execution to the next code fragment: either the next trigger, or if no more are specified, to the method specified in the method clause. Once these fragments are executed, '$$' transfers execution to the refinee operation at the supertype level.

Thus, in the case where only a base method is defined, '$$' functions exactly as 'super' in Smalltalk. However, when triggers are used, this is not the case. Consequently, VBASE avoids the super syntax in favor of the '$$' syntax to avoid the impression of moving up the supertype chain. Rather, '$$' simply transfers execution to the next code fragment, whatever that may be.

Functionally, '$$' behaves like a function call. Thus, the placement of the '$$' in the code allows implementation of pre-processing, post-processing, or both: wrapper processing.

One interesting subject regarding this implementation of method combination is the compatibility of operation specifications in a chain of operation refinements. In a strongly typed system such as VBASE this is an important issue. The approach we have taken focusses on guaranteeing a conformance relation.

Methods are checked by the COP compiler for conforming to their specification defined by the operation. Refinements of inherited specifications are verified by the TDL compiler for conforming to the original specifications. The specific criteria for conformance have been motivated by the work of Cardelli[CAR1984].

3.7 Exceptions

In many languages, there are no specific exception handling mechanisms. Thus code to detect and handle exceptions must be explicitly inserted at each point in a program where an exception might occur. This not only forces the writing of a great many short, repetitive code fragments, it also places an additional burden on the establishment of extra-language applications conventions and creates numerous opportunities for lapses in programming discipline.

In VBASE, we included a specific exception handling mechanism. Exception conditions detected during the execution of an operation raise an exception. That is, they transfer control to a pre-defined exception handling routine rather than return control to the caller.

Once again, referring to figure 3, note the except and raise statements. These statements allow graceful handling of abnormal events that occur during processing, and are variations on a fairly standard theme. What is notable is that in VBASE, exceptions are types. This means that all of the behavior definition mechanisms available to types are available to exceptions. One consequence is that the implementor can define a hierarchy of exceptions. Thus exceptions can be generalized just like types are generalized. For example, a memory allocation operation might raise the exception OutOfMemory. A refinement of the operation, say one which allocates memory for strings, might raise a more specific exception, say StringSpaceFull. StringSpaceFull could be implemented as a subtype of OutOfMemory. As a subtype of OutOfMemory, it could be used in any context where OutOfMemory itself would be expected.

The second implication of exceptions as types is that one can define properties and /or operations of exceptions. Properties can be extremely useful. In the previous example, one could add the property AmountRequested to the exception type OutOfMemory. For example, assume the raising routine returned:

```
raise OutOfMemory (AmountRequested: 4000000);
```

The 'catching' program could then issue a meaningful error message or do something else appropriate. For example:

```
except (o: OutOfMemory)
{
   printf ("The amount of memory: %d, requested is
not available\n", o.AmountRequested);
}
```

One can thus consider each actual raising of an exception as creating an instance of the exception. This instance is available to the catching program, which can treat it like any other object, accessing its properties, etc.

4. Interesting Database Aspects

VBASE supports most of the expected functionality of a DBMS. Objects can be shared among multiple processes concurrently, backup and recovery facilities are provided, and simple access control is available. A first version of an object query language is also under development for inclusion in the first version of VBASE. Beyond this, there are many notably different aspects of VBASE that derive from database influences.

4.1 Persistence

Persistence of objects is clearly the most notable difference between VBASE and most current object systems. Any time an object is created, either by a TDL definition or an invocation of a create operation in COP, it is considered permanent and continues to exist until it is explicitly deleted by a delete operation. The ability to deal with persistent objects without any special effort is an enormous advantage of VBASE.

4.2 Clustering

Another database influence apparent in the system is the ability to cluster objects on disk and in memory. Every create operation allows the invoker to specify a previously-existing clustering object. The new object is then clustered in the same segment as the clustering object. Since segments are the unit of transfer to and from secondary storage in VBASE, whenever any one of the objects in the cluster is accessed, the segment is transferred to memory (if it is not already there). Thus any subsequent references to one of the clustered objects will not require a disk access.

This has numerous applications. For instance, objects contained within an array can be clustered with the array. It is also very useful for a-part-of, or component, hierarchies, which are extremely common in engineering and text management applications. In this case, all the component objects can be clustered. Therefore only one disk access is required to transfer the entire hierarchy into memory. Clustering also provides space saving benefits, as there is less overhead when objects are stored in one segment.

4.3 Inverse Relationships

Reviewing the TDL definition of the type Part (figure 2) points out a further database influence in VBASE—the support for inverse relationships. Note the components and componentOf property definitions. These properties are declared as inverses. This means that whenever a modification is made to one of these properties, the other property is modified accordingly. This construct solves one of the more vexing problems in database management systems, particularly relational database systems. One-to-one, one-to-many, and many-to-many relationships between objects can all be supported and maintained automatically using the inverse capability. Thus, such common relationships as Parts–Suppliers or Employees–Departments can be implemented directly with no additional definitions or code. This is a dramatic improvement over most current database systems, and is not available in current object systems.

4.4 Protecting the Object Database from Process Failure

A last database style aspect of VBASE is the support of a minimal protection scheme. Current object systems are entirely memory resident and generally ignore the issue of corruption due to process failure. However, this has long been a standard issue of database systems since large amounts of important data are being manipulated. Some degree of safety and resilience must be offered. VBASE will offer concurrency control and recovery in its first release.

4.5 Triggers

The availability of triggers, discussed previously, can be considered both a language and a database influence. Many database systems talk about triggers, few implement them. Their utility is obvious. Triggers can be attached to properties as well as operations to generate whatever behavior is desired. These behaviors include standard ones such as 'when my QuantityOnHand property falls under twenty, issue a new order for a hundred more', to more esoteric patterns such as keeping audit trails of property and operation access for security purposes.

In VBASE the triggers are often used to augment creation and deletion methods. The use of triggers can insure, for instance that, upon creation of an object, all important referent objects are created as well. Delete triggers reverse this to delete all referent objects. Consider the example in figure 4.

When a PipeConnector object is created, one would also like to create an Array for the bolts property of the connector, perhaps initializing it from a set of Bolts passed to the Create operation. The use of a trigger on the standard create operation provides this functionality.

```
method obj PipeConnector
PipeConnector_CreateTrigger
    (aType, numberOfBolts, boltSet)

obj Type aType;/* must always take a Type arg when
    doing a create */
obj Integer numberOfBolts;
obj List[Bolt] boltSet;
{
  obj PipeConnector newConnector;  /* the result of
      the creation process */
  obj Bolt aBolt;        /* range variable for bolt set */
  int j = 0;             /* standard C variable */
  newConnector = $$ (aType);
    /* create the new object by invoking the
           standard system create operation */

  /* create the referent object */
  newConnector.boltSet =
    Array$Create ($Array, numberOfBolts);

  /* initialize the referent object */
  iterate (aBolt = boltSet)
    newConnector.bolts[j++] = aBolt;

  return (newConnector);
}
```

```
define Type PipeConnector
      supertypes = {Part};

      properties = {
        bolts: Array[Bolt];
        ...
      };
end PipeConnector;
```

Figure 4. Triggers can add behaviors to a create operation. The trigger method is shown above, the type definition is in the shaded block at right.

Two aspects of the system should be noted in passing. First is the arbitrary combination of C program variables with object variables. This, as stated, was an important goal: a truly integrated language. The language processor does all necessary conversion to assure a correct program is produced. The second factor is the iterate statement. Drawn from CLU, this statement processes all members of a database aggregate an element at a time without requiring the writing of a 'for' loop. This is yet another productivity gain of the system, as it is unnecessary to compute the boundaries for a for loop. Perhaps more importantly, iterators provide access to the elements of an aggregate abstractly, without exposing (or requiring knowlege of) the underlying implementation.

4.6 Access To Meta Level Information

The final attribute of VBASE drawn from DBMS's is the availability of meta information. VBASE is entirely self-describing: all system characteristics except the lowest layers of storage management are implemented using types. The properties and operations of these system types are freely available to programmers to use to their advantage. This makes system development easier, and allows implementors to create customized tools of their own while taking advantage of system tools already in existence.

5. Some Further Unique Aspects

5.1 Customized Property Implementations

Object systems are known for their ability to allow users to create customized abstractions. VBASE provides users with the unique ability to customize implementations as well. This ability is available at two levels.

In the simpler case, an implementor can provide customized access to a property by replacing the default get and set operations for the property by customized ones. For instance, the property 'age' in the following example has such customized operations specified.

```
define Type Person
  supertypes = {Mammal};
  properties = {
    age: Integer define set
        method (Person_SetAge)
         define get
        method (Person_GetAge);
  };
end Person;
```

This specification will cause the user defined routines to be invoked whenever access to the age property occurs as in:

```
{
obj Person aPerson;
obj Integer theAge;
....
theAge = aPerson.age;
....
}
```

What is different here from most systems is that when both a get and set operation are specified, no storage is allocated. Thus the programmer truly takes over the implementation, including storage allocation. The user may choose to calculate the value (in the case of age, it is common to calculate the value as the difference between the person's birthdate and the current system date), in which case no storage is needed. If storage is necessary, the implementor may allocate it wherever he/she desires. For example, in a design application one might store large bitmap graphic images using a compression algorithm, and write customized code to read and write the image. In a similar vein, in a CASE system one might store fragments of source code in standard operating system files so that the various language processors will recognize the fragments. Finally, data from alien databases can be imported and exported transparently by using customized properties. The get and set operations are used to call the appropriate database routines on the foreign database to read and write the data.

5.2 Customized Type Implementations - MasterTypes

The use of custom routines for handling property implementation still incurs the overhead of a standard object. There is space overhead for the default representation, and the overhead of the system routines for dispatching to the user's custom routines. For sophisticated users wishing to avoid even this overhead, VBASE allows the complete implementation of customized types. Since VBASE provides the complete specification of all system types including type Type, a complete customization of a type is possible. It requires substantially more work than a custom property, but this is to be expected.

Customized types are actually handled as a subtype of type Type, called MasterType. The most significant characteristic of a MasterType is that it takes over the dereferencing operation. VBASE insists on strong reference semantics. That is, objects are always represented by a reference, and these references appear uniform from the outside. Thus, the 'every object is a first class object' semantics is maintained; even integers, single characters, and booleans are true first class objects. However, the types Integer, Character and Boolean are also MasterTypes. They implement their own creation, deletion, and dereferencing operations. This allows types such as Integer to store their value within their reference, and for types such as Real to make use of special hardware to implement arithmetic operations. MasterTypes must implement a create routine which fabricates and returns a reference, an appropriate dereferencing routine, routines for property access and operation dispatching and invocation, etc. However, once the complete specification has been met, these MasterTypes behave exactly like all other types to users, and all of the attributes of the VBASE environment can be used with them.

Implementors can therefore use the MasterType feature to create extremely customized types. No space or time overhead is incurred because the user implementation handles everything. MasterTypes are very useful for implementing custom access methods which require special data formats. This is a unique aspect of VBASE: the ability to tune access to special data formats such as large blocks of text or graphics, while remaining within the basic system.

Another use for MasterType implementations is the construction of efficient integration databases. A model of a complex data structure is created through the definition of the appropriate types, properties and operations. This data, which is actually stored in existing foreign databases, is accessed through MasterTypes which transfer the data to and from the alien databases. The use of MasterTypes allows a relatively efficient interface to the foreign database system to be implemented, while the processing and data modeling can be done in VBASE, with the attendant increase in modeling power and ease of implementation.

5.3 Free Operations

VBASE defines free operations: operations that are not associated with a type, and consequently, are not invoked via the standard dispatching means. In object message systems, every message is dispatched; that is, the type of the object being sent the message is used to find the method which implements the message. Free operations in VBASE, in contrast, do not have a distinguished argument. They are simply procedures free of type association.

Summary

VBASE has, we hope, achieved all of the goals we set for ourselves, at least to some extent. It is a relatively complete development system with language processors and development tools. It is object based, strongly typed, and provides support for persistent objects. It also allows custom implementations for improved efficiency. VBASE contains many interesting features from both the language and database spheres. In fact, the most interesting aspect of VBASE is that it cannot be strictly classified as a language or a database system.

References

BOR1982 Borning, Alan H. and Ingalls, Daniel H. H.; "A Type Declaration and Inference System for Smalltalk"; Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, pp 133-139, 1982.

CAR1984 Cardelli, Luca; "A Semantics of Multiple Inheritance"; Lecture Notes in Computer Science. Springer-Verlag, New York, 1984, pp 51-67.

CAR1986 Cardelli, Luca and Wegner, Peter; "On Understanding Types, Data Abstraction, and Polymorphism"; Computing Surveys, Vol. 17, No. 4, December 1985

CAT1983 Cattell, R.G.G.; "Design and Implementation of a Relationship-Entity-Datum Data Model"; Xerox Corporation, 1983

COP1984 Copeland, George and Maier, David; "Making Smalltalk a Database System;" Sigmod '84, Sigmond Record Volume 14, Number 2, pp 316-324, Association for Computing Machinery, 1984

COX1986 Cox, Brad J.; *Object-Oriented Programming: An Evolutionary Approach;* Addison-Wesley, Reading, MA 1986

GEM1986 *GemStone Product Overview, GemStone Version 1.0;* Servio Logic Development Corporation. March, 1986

GOL1983 Goldberg, Adele and Robson, David; *Smalltalk-80: The Language and its Inplementation;* Addison- Wesley, Reading, MA, 1983

JOH1986 Johnson, Ralph; "Type Checking Smalltalk"; in Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications; pp 315-321; Portland, OR; September 29-October 2, 1986

LIS1981 Liskov, Barbara; Atkinson, Russell; Bloom, Toby; Moss, Eliot; Schaffert, J. Craig; Scheifler, Robert and Snyder, Alan; *Lecture Notes in Computer Science;* Springer-Verlag, New York, NY 1981

MEY1986 Meyer, Bertrand; "Genericity versus inheritance"; in *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications;* pp 391-405, Portland, OR; September 29-October 2, 1986

MEY1987 Meyer, Bertrand; "Eiffel: Programming for Reusability and Extendibility;" SIGPLAN, Notices, vol 22, no 2, pp 85-94; February 1987

MYLO1980 Mylopoulos, John, Bernstein; Philip A., and Wong Harry K.T.; "A Language Facility for Designing Database-Intensive Applications", *Transactions on Database Systems;* Vol 5, No 2. pp 185-207; Association for Computing Machinery; June, 1980

STON1986 Stonebraker, Michael and Rowe, Lawrence A. "The Design of Postgres"; Sigmond Record, vol 15, no. 2, pp 340-355; Association for Computing Machinery; June 1986

STR1986 Stroustrup, Bjarne L.; *The C++ Programming Language;* Addison-Wesley; Reading, MA, 1986

TES1985 Tesler, Larry; "Object Pascal Report"; Structured Langauage World, vol 9, no.3, 1985