Cognitive Analysis, Design and Programming: Next Generation OO Paradigm*

JOSEPH M. SCANDURA
Merger Research Institute, University of Pennsylvania and Scandura-Flexsoft International

(Revised September 1996)

The object-oriented (OO) paradigm has clearly come of age. There is no question as to whether OO will replace traditional approaches to software development. It is just a question of how quickly one should move to OO, and to a lesser extent how "pure" one should be. Just as structured programming eliminates "spaghetti" code, for example, OO makes it possible to model real world objects. It also strongly encourages code reuse. Although OO design methods are still evolving (e.g., Bruno, 1995), many of the better known OO methodology authors (e.g., Booch, 1994; Rumbaugh et al., 1991; Coad and Yourdon, 1991) are in general agreement as to how software systems should be designed and implemented (Orfali, 1994; Palmer, 1994; Rogers, 1994; Wood, 1994).

In spite of its advantages (e.g., improved modularity and reusability), however, the OO paradigm has a number of important weaknesses. Perhaps most fundamental is the problem of how to combine objects in producing large systems. Mutual dependencies complicate flow of control making it difficult to debug complex OO applications. Fixing problems can be even more complex. System behavior is easily obscured in message passing and unconstrained inheritance.

*The author wishes to thank Dr. Mort Hyman, Christian Simms, Thomas Honigmann and several unnamed reviewers for their helpful comments on a draft of this article. The author is fully responsible for remaining inadequacies.
Viewing “objects” as both star and supporting cast has the effect of reducing the importance of operations. Although organizing functions by data type has certain advantages, it also has disadvantages. Operations having little conceptually to do with one another are, nonetheless, in practice often grouped together.

In the real world, actions play as important a role as objects. At a minimum, one needs both “nouns” and “verbs”. One cannot express a complete thought (i.e., write a sentence) without using or implying both an object (noun) and an action (verb). At first thought, it might appear that actions in OO are adequately handled by functions within objects. But this is not the case. Verbs act on nouns as indivisible wholes. Functions within objects are clearly subordinate (to objects); they act more like modifiers (adjectives?).

As systems become increasingly complex, OO methods are no different than those which came before. In each case, the number of tests required for verification goes up exponentially with complexity (Scandura, 1992). Consequently, the traditional design-implement-test paradigm – whether OO or otherwise – is inadequate as regards verification. Making it possible to test designs at higher levels of abstraction helps reduce testing to manageable proportions. The ability to ensure logical consistency of models as they are being developed, of course, would be even better.

**Purpose.** In this paper, I introduce a cognitive OO approach to software design and programming based directly on “real world” semantics. Specifically, I show how this cognitive approach together with existing Flexsys simulation technology: (a) makes it possible to model “real world” behavior more directly and more fully than the standard OO paradigm, and unlike traditional structured design to test and debug these models as they are being designed, (b) directly supports (as opposed to enables) abstract input–output operations as well as objects, and as such is a natural successor to OO in the evolution of programming paradigms, (c) can be implemented in C++ (and other OO languages such as Ada 95), and (d) supports code reuse, including the reuse of legacy code. I refer to this cognitive approach as the Cognitive Object Oriented (COO) paradigm.

Reference in the paper is made to Flexsys, an integrated software engineering system which supports many of the ideas introduced. In a

---

1Except where used in connection with C (or Ada), the term “function” is used here in a generic sense to refer to any operation. The terms structural (cognitive task) analysis, cognitive analysis and cognitive OO (COO) are used largely interchangeably throughout.
separate article, I show how the COO approach, together with Flexsys software technology, also makes it possible to guarantee that software is correct by design (patent pending).

**COGNITIVE OO DESIGN**

The cognitive OO paradigm has been motivated by earlier work in analyzing cognitive processes, specifically by a method I originally called Structural (cognitive task) Analysis (Scandura, 1971; 1977; 1982; 1984). (NOTE: The “al” at the end of “structural” is intended since the work was quite independent of Wirth’s (1976) contemporaneous structured programming and structured analysis and design which came later.)

All systems involve input, output (which may be the same) and operations which map the first onto the second. The essence of the cognitive OO paradigm (alias COO/structural analysis) involves modeling system behavior successively from the highest levels of abstraction until contact is made with available data and computational resources. Artificial barriers between high and low level design and programming are eliminated.

At the top level of analysis, all system modeling begins by labeling the abstract input, output and operation. In modeling an event driven system, for example, we initially refer to the incoming events collectively by assigning them a generic but descriptive label. For example, we might refer to Flexsys (key presses and mouse events) where “Flexsys” is the name of a complex software engineering application. Similarly, the initial output refers to the collective outputs of the application – say “software engineering outputs”. The top level operation, then, might be described as “software engineering operations”.

Clearly, at this level of abstraction, little is known about the application other than its domain of applicability, and then only in a very generic sense. Notice, however, that abstract inputs (i.e., “Flexsys events”) and outputs correspond to virtual objects, much like a base class in C++ or a top level package in Ada 95. Indeed, we will find operations lurking inside the objects as we probe more deeply. The abstract operation “software engineering operations”, however, is different. It acts on and produces objects as wholes, with only implicit reference to specific procedures or functions contained in objects (or equivalently to passing “messages” to the objects).

In modeling behavior, structural (cognitive) analysis shares much in common with structured analysis. Parallelizing structured analysis, for
example, operations can be refined into a sequence, a selection (e.g., IF–THEN) or iteration (e.g., LOOP). One can also envisage parallel execution. In structured analysis, however, data is not refined (or even considered) as such. It is simply assumed to be available as needed.

In contrast, data refinement is an explicit part of cognitive analysis. Abstract data must be refined in parallel with abstract operations. We return to this matter after discussing process refinement.

**Process Refinement.** If a process is refined into a sequence, then the abstract input and output remain as before. If the refinement is a selection, however, we must introduce an "extra domain" condition variable or relation which partitions the domain. Domain elements in one equivalence class of the partition serve as input to the THEN statement. Inputs that fall in the other equivalence class of the partition are input to the ELSE statement. Clearly, selections can be generalized to CASE structures involving any number of elements in the partition. Since the output of the original abstract operation must reside in the abstract range variable it follows that the output of each CASE alternative must also lie in that same range variable.

The case of iteration can be viewed as a variation on binary selection in which the THEN statement, or LOOP body, accepts inputs in both partition elements. In turn, outputs of the LOOP body must be legal values of the abstract domain variable with elements in both equivalence classes of the partition. The hard part is to ensure that each iteration of the LOOP body modifies outputs in a systematic way so as to ensure a transition between the equivalence classes in the domain partition (one of which exits the loop).

One might also refine an operation into two or more operations which act in parallel on the domain and collectively produce elements in the range.

**Data Refinement.** Except for terminology and perhaps emphasis, cognitive analysis so far appears identical to structured analysis. Structured analysis presumes concrete inputs and outputs. In contrast, every cognitive analysis begins with a virtual input, output and process.

Abstract inputs clearly do not provide the detail necessary to reflect the data complexity required for implementation. Data structures are introduced gradually in parallel with process decomposition. The equivalence classes introduced by a selection partition, for example, correspond to new domain variables, in particular to subsets of the original abstract domain. Abstract inputs and outputs become increasingly concrete as a result of refinement, providing more and more
information about the data structures involved. Eventually, refinement of virtual data makes contact with real data.

What is essential in cognitive analysis is that data and process remain in sync at each level of refinement. This is true whether talking about virtual data and processes or real ones. (We see below that this coordination of data and process has important implications for testing and debugging design models.)

Data refinement leads naturally to hierarchical data structures, analogous to record structures. (Arrays in this context are simply special cases.) Other kinds of relationships also arise naturally as a result of the “extra domain” conditions introduced to partition abstract data elements. In particular, conditions may be defined as functions on other data. Moreover, concrete data structures may be derived from abstractions – much as in inheritance. Conversely, abstract data may be derived in terms of the more concrete – often called “evaluation”. Defining conditions in terms of other data comes up naturally in the following example.

**Example.** Irrespective of the system being modeled, cognitive analysis always begins with INPUT, OUTPUT and OPERATION (see Fig. 1A). For illustrative purposes, consider the simple task of cleaning a room. (The example might be suggestive if you have young children at home, or are into robotics.)

The first step is to name the top level INPUT, OUTPUT and OPERATION abstractions (see Fig. 1B). Notice that the top level abstraction under OPERATION is a full statement referencing both “room” (which in this example serves as both input and output) and the operation “clean”. The terminology used in this simple High Level Design (HLD) language is completely arbitrary and can be chosen as desired to reflect the underlying semantics.

**Simulation.** Our Flexsys software includes an HLD Simulator which supports dynamic “execution” of HLD designs. High level

```plaintext
{FIGURE_1.FLX}:name_of_model;[]library:procedure;

[INPUT]:

[OUTPUT]:

OPERATION
```

**Figure 1A.** The common starting point for modeling any system. “Figure_1.FLX” is the name of the Flexform file, “name_of_model” is the model name, “library” is the name of our extensible high level design language and “procedure” indicates the type of Flexform. Other symbols are strictly syntactic and may be ignored.
Figure 1B. A top level model of “clean room”. In this example, “room” is a clone, marked with “@”. There is only one room under both INPUT and OUTPUT. Room initially refers to the state of the system at the beginning of execution. (After execution the value is “presentable”.)

designs can be executed interpretively at any level of abstraction. Test coverage and various debugging modes also are supported. Execution is automatic where referenced components have been semantically wrapped and integrated into the Flexsys software.† Virtual (abstract) components are simulated manually. The user is shown the current status of the data and is asked to perform the virtual operation and update the data. Both real and virtual components interoperate so designs can be tested (and debugged) at successive levels of detail through implementation. In the process, elements in one design module can call other modules, with explicit support for recursion.

In particular, the high level clean room specification can be run as represented in the HLD Simulator. At the beginning of execution, the value of “room” is “unpresentable” as shown. Execution of this high level model involves only the operation “clean”. During execution, the cursor stops at the operation, the current value of “room” is displayed (i.e., “unpresentable”) and the user is asked to perform (simulate) the “clean” operation and update the value of room. In this case we might enter “presentable”.

Sequence Refinement. In Fig. 2, “room” is refined into, and defined in terms of, the components “bed” and “carpet”. As detailed below, the value of room is a function of bed and carpet. Correspondingly, the behavioral design is refined into a (possibly parallel) sequence of two operations: make the bed and vacuum the carpet. Initially, the bed

† Contact jms@pobox.upenn.edu for more details.
Figure 2. Sequence refinement of “clean (room)” into “make (bed)” followed by “vacuum (carpet)”.

Figure 3. Illustrates refinement into a loop.

is unmade and the carpet is dirty. After execution, the bed might be made and the carpet, clean.

Loop Refinement. Figure 3 shows what happens when we refine “vacuum (carpet)” into a loop. In this case, we “advance_and_vacuum” the carpet in increments: First “seg1”, then “seg 2” and finally
"seg3". In effect, refinement of the operation "vacuum" (in the statement "vacuum (carpet)") into a loop requires that carpet be partitioned into segments. Notice that "carpet" has been defined as the "union" of three segments - a classic case of evaluation. The value of carpet is a function defined on values of the three rug segments. If all of them are clean then the carpet is clean. Otherwise, carpet is dirty. Also notice that the loop requires introduction of an alias (a pointer), providing a common reference to the various segments (one at a time).

**Loop Simulation.** In this case, execution begins as before with "make (bed)". Then, the user is told to "advance_and_vacuum" the current_segment. Current_segment initially has no value. Since it is an alias, however, the HLD Simulator knows it must refer to some element. When asked, the user might supply "seg1". Once the meaning of current_segment is known, the user is shown seg1's current value and is asked to update its value (after vacuuming). In short, the statement

\[
\text{advance\_and\_vacuum (current\_segment)}
\]

involves first determining what is being aliased, and then updating the value of the referenced element (e.g., seg1).

The loop condition is simply a matter of evaluating "done". "Done" is an example of what we called earlier an "extra domain" condition. It partitions the carpet segments into two equivalence classes, one which includes segments (seg1, seg2) and returns control to the loop body and one (seg3) which exits the loop. Notice also that the value of "done" is determined by a function defined on existing elements (the alias and seg3). Specifically, this function (same) determines whether current_segment is an alias for "seg 3". One must execute the loop body three times before the condition is true.

Incidentally, "same" is just one of the semantic core functions built into the Simulator which provide a full set of semantic model building facilities. Execution of "done" in this case is automatic.

**Further Refinement.** Figure 4 shows further refinement of the clean room model. In this case the loop body has been refined first into a sequence, then the second operation into a selection and finally the THEN alternative into a sequence. The first statement in the body

\[
\text{current\_segment} = \text{next\_element (carpet, current\_segment)}
\]

involves "next_element", another of the core semantic functions. Each time through the loop, this statement automatically assigns the next segment to current_segment. The condition "match" checks to see if
current_segment is an alias for a segment that is "messy & dirty". (Seg 2 is the only segment with this value.) In effect, the selection refinement partitions the segments into two equivalence classes, those that are "messy & dirty" and those that are not. When the THEN sequence is executed, the "pickup" operation changes the value of current_segment to just "dirty" – that is, it gets rid of the "messy" part. The operation "vacuum" does the rest (i.e., it cleans seg 2).

Data-Process Abstraction and Design Debugging. Notice the natural progression inherent in the process. Abstract operations and objects can be refined successively and indefinitely. There is no natural or required transition between high level and detailed design. Indeed, as noted below, there is no sharp distinction between detailed design and programming, other perhaps than shifting from a generic representation
to a specific one (e.g., to a programming language like C++ or Ada 95). Successive phases of refinement flow naturally from one to the other. Any divisions introduced between high level and detailed design are arbitrary and strictly for convenience.

The fact that data (objects) and process (operations) must be represented at the same compatible level of abstraction has important implications for testing—in two words “design debugging”. Abstract objects are operated on by correspondingly abstract operations. Consequently, the HLD Simulator (above) can execute any such model at arbitrary levels of abstraction. This makes it possible to simulate and debug systems as they are being developed. (We shall see below how data refinement relates to OO inheritance.)

Scandura has shown elsewhere (1990, 1992) that debugging designs from the highest levels of abstraction dramatically reduces the number of empirical tests required to “prove” a system. When testing is delayed until after a system has been implemented, the number of paths to be tested goes up exponentially with complexity. This number only goes up additively when testing is done successively from the highest levels of abstraction.

To summarize, the major difference between the above and traditional structured analysis and information modeling is that structural (cognitive task) analysis requires parallel refinement of both processes and data. Unlike structured analysis, process refinement (sequence, selection, iteration) imposes parallel requirements on virtual data. In addition to simple decomposition, data objects (elements) and their values can be defined in terms of functions on other elements.

Notice that the same (or equivalent) behavior is determined irrespective of the level of data and/or process abstraction used in a simulation or execution. For example, “clean (room)” results in the same behavior—a presentable room—as does “make (bed)” and “vacuum (carpet)”. A room is defined to be unpresentable when either the bed is unmade or the carpet is not clean, and presentable when the bed is made and the carpet is clean. Given this functional relationship, the parent operation, clean, performs the same behavior as do the subordinate operations make and vacuum. In effect, maintaining compatibility between data and process at successive levels of refinement makes it possible to test and debug software as it is being designed—in a manner which is far more efficient than the traditional design-implementation-test paradigm (patent pending).

As we have seen, data objects (and their values) can be constructed dynamically by functions defined on other objects.
Extension to Object Inheritance. So far, we have assumed that the virtual components used in designs, as well as components used to implement designs, are defined on data structures composed of indivisible objects (also called variables). We also have seen how objects in such structures can be defined in terms of other objects. No behavior, however, was ascribed to the data structures themselves. The component objects (e.g., “room”, “bed”, “carpet”) comprising these structures were assumed to be elementary with no internal structure as such.

Inheritance in the OO paradigm involves relationships between categories of objects rather than between objects and their components. Category relationships typically involve reference to internal structures. A bedroom, for example, is a category of room. It is not a component. The internal structure of room may or may not be made explicit (e.g., it might be defined in terms of walls, floor and ceiling). To be a bedroom in either case requires additional structure (e.g., beds and carpets). Categories of rooms, say bedrooms and kitchens, necessarily include both the components of rooms in general and the components which distinguish those categories. Thus, bedrooms typically contain beds and carpets. Similarly, kitchens contain stoves and sinks.

In traditional OO, rooms and bedrooms correspond to separate, albeit hierarchically related classes (types with associated functions defined on those types). In the real world, however, objects do not correspond directly to such classes. Rather, bedrooms for example include the components associated with both room and bedroom classes. As a result, OO terminology can be misleading and a source of conceptual errors in design.

Similarly, when we speak of cleaning rooms from a real-world semantic (cognitive) perspective, reference is made to all rooms. Making beds and vacuuming carpets are integral parts of the process when the room in question happens to be a bedroom. But, this level of procedural detail (in a design) is not necessary unless we are talking explicitly about beds and carpets. When we say “clean (room)”, and the input room is a bedroom, the intent is to make the bed and vacuum the carpet, as well as to clean the generic parts of the room. Real-world cognitive semantics calls for an abstract, albeit executable operation which operates on all rooms. Such a clean operation implicitly includes make and vacuum.

This raises the question of how best to implement a procedural statement like “clean (room)” when functions like make and vacuum (in subordinate objects) are affiliated with clean (a function in a superordinate object). In most OO languages, such affiliation is most commonly handled by introducing overloaded clean operations. In this
case, when a room object is passed to clean, the program must decide which clean to invoke. The choice is typically determined by the type of the room object. In typeless OO languages, the same result can be obtained by choosing the function whose domain most fully matches the object's structure. The clean associated with the bedroom object would be chosen when the room in "clean (room)" is a bedroom – because it offers a better fit than does the original clean (i.e., it deals with beds and carpets as well as generic room components).

The latter clean operation might be implemented, for example, by introducing case alternatives (within the operation) which anticipates everything that rooms might contain. Alternatively, the case alternatives might refer to the kinds of clean sub-operations which need to be performed (e.g., on objects having hard impervious services, on cloth-like items to be arranged symmetrically, etc.).

The same results can be accomplished in a way that more naturally reflects real-world semantics – by introducing the notion of affiliate functions. In modeling system behavior at a high level of abstraction, clean is assumed to operate on all room objects. From a real-world behavioral perspective, it is an external operation (and not just one of potentially many functions associated with room objects). "Clean (room)" means: Clean all of the components which define room (e.g., walls and floor) plus all of its contents (e.g., beds, carpets or whatever).

This affiliation concept can be implemented directly by associating functions in subordinate objects with operations in (or operating on) superordinate ones. When we add a function, like make or vacuum, to a subordinate object, we would be given the option of affiliating that function with functions associated with the superordinate object. In this case, calling "clean (room)" would have the effect of (also) executing those of its affiliates that are associated with any given room object.

The latter implementation, of course, might retain the option of specifying functions associated with particular objects – perhaps using the common syntax: object.function (data). For example, "bedroom .clean (room)" could be used to specify a particular instance of an overloaded clean function. (NOTE: From a behavioral perspective, there is little difference between calling a function in a specified object and calling one in a function library.) The latter syntax might be deemed essential by many OO programmers. Nonetheless, I suspect, but have not attempted to prove, that this kind of flexibility is unnecessary. Anything that can be done using the extended syntax can also be done in the proposed semantically more palatable manner – much like GOTOs can be eliminated in structured programming.
EVOLUTION OF PROGRAMMING PARADIGMS — FROM OO TO COGNITIVE OO PROGRAMMING

In his defining book on C++, Stroustrup (1991) makes a useful distinction between enabling a paradigm and supporting one. A language supports a programming style if language facilities make it convenient, "reasonably easy, safe and efficient" to use that style. If it takes exceptional effort or skill to write programs in a given style, then that language merely "enables" the technique. As an example Stroustrup observes that one can write structured programs in FORTRAN and object-oriented programs in C, but it is difficult to do so because these languages do not directly support those paradigms.

Evolution of Programming Paradigms. A series of programming paradigms have been introduced over the years, each making it easier than its predecessor to develop, understand and maintain larger programs. Programs originally consisted of single monolithic programs operating on single sets of variables.

As monolithic programs got bigger and more complex, they became harder and harder to understand. To overcome this limitation, programs were generalized to include subroutines (see Fig. 5).

Procedural Programming Paradigm. The procedural paradigm supports breaking large programs into subroutines.

A major problem with procedural programming is that many (global) variables retain their values across subroutines. This dispersal frequently leads to unwanted and/or unpredictable side effects, making programs harder to test and debug. To overcome this limitation, design constraints were imposed on the use of global variables.

Modular Programming Paradigm. Modular programming represents a generalization of procedural programming involving multiple compilation units. Here the emphasis shifts from the design of procedures to the organization of data. The modular paradigm restricts global effects to modules. In addition, an explicit interface is defined to govern calling relationships between routines in different modules.

Routines in given modules, however, frequently have little or no relationship to one another. This distorts the relationship between program structure and the program's "real world" semantics. Allowing unrelated routines in the same modules and/or related routines in different modules makes it difficult for end users (and often designers) and programmers to communicate. To overcome this limitation, design constraints were imposed as to which routines should go in given modules.
Evolution of Programming Paradigms

**Procedural Program**

```plaintext
fn1 (var1, ...)
fn2 (var2, ...)
...
```

**Modular Program**

```plaintext
fnA1 (varA1, ...)
...
```

**Data Abstraction Program**

```plaintext
fnA1 (varA1, ...)

type A
...
```

**Object Oriented (OO) Program**

```plaintext
fnA1 (varA1, ...)

type A
...
```

**Figure 5.** Schematic representation of procedural, modular, data abstraction and object-oriented paradigms. Modular programming involves both generalization of procedural programming to support multiple compilation units and restrictions on data usage in modules and/or units. Data abstraction largely constrains modular programming by grouping functions by data types. Object orientation generalizes data abstraction *via* inheritance.
Data Abstraction Programming Paradigm. Data abstraction organizes (restricts) routines in modules according to the data types used (e.g., C++ Classes, Ada Packages). The basic paradigm involves deciding which types of data are needed and providing full sets of operations for each such type. Data abstraction tends to go along with greater use of parameter passing. Modules in this case correspond to objects in the “real world”. Generic modules also are often supported (e.g., Generic Packages in Ada 83).

Abstract types are routinely defined in terms of simpler ones. Abstract data types in this paradigm, however, cannot accommodate new uses without explicitly modifying their definitions. Every new module (object) must be developed from scratch even where data structures are very similar to ones which have already been developed. To overcome this limitation, the data abstraction paradigm was generalized to allow inheritance between modules and function polymorphism.

Object Oriented (OO) Programming Paradigm. The object-oriented paradigm allows new classes to “inherit” from existing classes (the OO name for modules). The latter then are referred to as superordinate classes. Functions in those superordinate classes are automatically defined on the subordinate classes. In addition, conceptually similar functions in different classes may be overloaded (i.e., use the same name but operate on different data structures). Polymorphism ensures that the appropriate function is invoked dynamically at run time based on object types. Procedure calls between objects are accomplished by “message passing” (the OO equivalent to procedure calling). Data structures in classes are often but not necessarily defined statically. (They also may be derived from generic class/package definitions – as in Ada generic packages and C++ template classes.)

Limitations of OO Programming. In OO programming, inheritance refers to classes one way and to objects in another. Subtypes in subordinate classes supplement superordinate ones. Objects, on the other hand, include data structures associated with both superordinate and subordinate classes. For example, whereas a bedroom class refers only to structures unique to bedrooms, (bedroom) objects associated with that class include structures associated with both bedroom and room classes. This can easily lead to confusion or unnecessary complication in modeling real-world semantics.

Similarly, method (function) overloading (and/or polymorphism) typically involves distinct functions having the same names in different classes. Real-world semantics, on the other hand, typically calls for single all-encompassing operations which combine the effects of all
(overloaded/polymorphic) functions sharing a name. Behaviorally speaking, such all-encompassing operations operate on objects as wholes. Putting them inside objects is counterintuitive. By differing from this simple semantics, standard OO implementations lead at a minimum to unnecessarily complicated and hard to maintain code. Rather than simply writing one clean operation (with affiliates), which refers to all rooms (bedrooms and otherwise), for example, one would have to maintain several clean operations, one for each category of room.

An important difference between objects as resources and the use of objects in modeling behavior is also often neglected. Collecting functions inside objects because they use a common data type is reasonable from the perspective of defining resources. In modeling behavior, however, operations become paramount. In this case, it is cognitively more palatable to view different (albeit hierarchically related) data types according to the operations that use them.

Overall, semantic differences—between classes used in inheritance hierarchies and objects associated with those hierarchies and between overloaded/polymorphic operations and cognitive intent—lead to unnecessary complications in modeling behavior. Reliability, availability (understandability) and maintainability (RAM) are reduced as a result. Designs and programs become unnecessarily complex; debugging and testing can become even harder. To overcome these limitations, abstract all-encompassing operations, which incorporate affiliate operations, are introduced. These operations act on objects as wholes. Functions within objects may be referenced by (i.e., become affiliates of) these abstract operations but they are conceptually different. Other functions serve only to define relationships between objects via data execution (see Fig. 6).

**Cognitive Object Oriented (COO) Programming Paradigm.** The COO paradigm supports abstract (albeit executable) operations which operate on classes of input objects and which generate output objects. COO operations are based directly on real-world semantics. They accomplish their work by referencing affiliated functions associated with classes in inheritance hierarchies (used in defining object parameters). Although frequently hierarchical in nature, the parameters in Flexsys are typeless.

In addition to functions within objects, COO supports associating functions directly with objects. The latter serve to define given objects (dynamically) in terms of other objects, much like functions embedded in procedure calls. The only difference is that these functions are associated directly with data, rather than (necessarily) in procedural
logic. We call this “data execution”. Data execution simplifies procedural logic, and localizes the definition of new objects. Notice in our original clean example (Fig. 2) that room is defined directly in terms of bed and carpet. That is, the value of room is determined by the values of bed and carpet. Similarly, in the designing an algorithm for simulating the way a child performs column subtraction, data execution might be used to define, say, top-digit as the common element of top-row and current-column. Instead of having to define top-digit repeatedly in writing the algorithm, data execution makes it possible to do it just once.

**COO Analogy to Cognition.** The distinction in COO programming between abstract operations (which act on objects as a whole) and functions used to define objects has direct parallels in human cognitive processing. Functions associated with objects correspond rather directly to automated (well-learned, perceptual) processes, things which take place more or less automatically without conscious thought. In
the clean room example, one knows when one is "done". The functional definition is executed automatically as required.

More generally, functions effectively define component relationships between objects in a hierarchy. Data execution allows objects (e.g., rooms) to be defined dynamically in terms of other objects (e.g., beds and carpets). Data execution in COO is a modifier, serving a purpose analogous to adjectives (which modify nouns).

Operations between input and output objects refer to more conscious cognitive processing. In the real world we act on objects: We write, speak, shave, drive and eat. We do not just pass "messages" to objects telling them how to behave. Writing programs that reflect the way we naturally describe behavior has obvious advantages as regards both construction and understandability.

In effect, the COO paradigm provides a more complete characterization of the "real world", including actions as well as objects. This is analogous to a sentence having both a noun and a verb. (Functions defining objects are analogous to adjectives modifying nouns.) Cognitive programs are a natural consequence of COO analysis and design, and may be tested and debugged at arbitrary levels of refinement from the highest levels of design through implementation. A key concept is that the collective behavior of child operations in every refinement must be equivalent to that of the parent. Demonstrating equivalence in this case requires reference to functions defining relationships between parent and child objects, as well as between inputs and outputs.

**COO Design-Programming Continuum.** As shown in the clean room example, COO programming is a natural extension of COO design. Like COO design, COO programming directly reflects the way people describe the real world. The contents of Flexform elements, as well as overall Flexform relationships (both between Flexform elements and between different Flexforms – cf. Scandura, 1987; 1990; 1992; 1994), are of the same form whether one is talking about design or programming. The only difference is that terminal elements in Flexforms at the implementation level correspond to available resources – to data structures, functions and/or operations that are or can automatically be converted into machine readable form. These resources may correspond to native statements in a programming language, operating system services, or functions or procedures in a library.

**More Complete Representation of the "Real World".** The above discussion led naturally to abstract objects, abstract operations and (automatic) functions between objects. Nouns, verbs and adjectives were shown to be real world analogues. This raises the question of
adverbs. Is there anything in COO programming that corresponds to verb modifiers (i.e., adverbs)?

The answer is yes! As noted above, the COO paradigm derives from earlier research in structural (cognitive task) analysis of problem domains. Indeed, it is a short step from structural analysis of problem domains to domain analysis in software engineering. Structural analysis (e.g. Scandura, 1971; 1973; 1982; 1984) is a recursive process. The first step identifies sets of individual solution rules (alias programs) associated with the domain.

In human problem solving, the domain may be so large, however, that it is difficult to devise any finite set of programs which provide solutions for all problems in the domain. Structural analysis was originally designed to handle this situation (e.g., Scandura, 1971; 1973; 1984). Given an initial (lower order) set of solution rules, structural analysis builds on this base and offers a recursive method for identifying higher order rules (alias higher order operations/programs).

These higher order rules take (other) rules as input and generate rules as outputs. The input/output rules are of lower order (relative to the higher order rules). The important point is that higher and lower order rules operating collectively can generate solutions for (often much) broader ranges of problems than those same rules used individually. The power of the approach derives from often dramatic increases in problem solving potential at successive levels of structural analysis (Scandura et al., 1974; Scandura, 1977). As you may have guessed, higher order operations effectively modify other operations—analogous to "adverbs" modifying "verbs".

In effect, just as data execution supports dynamic modification of objects (i.e., data types), higher order operations support dynamic modification of operations. This latter idea has been frowned on in traditional programming because undisciplined, dynamic modification of operations often results in hard to detect side effects. When used in the highly disciplined manner, however, cognitive research strongly suggests that this type of "higher order" programming could add a degree of controlled power rivaling the typically more idiosyncratic use of AI techniques. The use of higher order components has much in common with the notion of self-modifying intelligent software agents.

Although COO technology has already been used to develop real industrial strength programs (see below), extensive application of higher order cognitive programming remains for the future.
IMPLEMENTATION OF COGNITIVE OO PROGRAMMING IN C++

In cognitive OO (COO) design all operations operate on parameters. (There are no global variables as such.) These parameters may take objects having different types as values. The objects themselves may or may not contain functions. In addition, functions may serve to define objects dynamically in terms of other objects.

It is of some interest to see how COO designs may be implemented in C++. Accomplishing this requires two things:

(a) a way to define elements of one type, or their values, in terms of elements of other types and
(b) a way to accommodate operations which operate on different types of elements.

(a) COO programming allows “real world” objects to get their values from, even to dynamically be constructed from, other objects. The value of room, for example, was defined in terms of a function defined on bed and carpet.

COO also supports the dynamic definition of object data structures (types) in terms of the types of other objects. That is, data structures can be dynamically constructed from other structures. For example, bedroom could have been defined in terms of room, bed and carpet structures. In COO, the term data execution refers both to deriving values of objects (variables) from functions of other objects and to deriving types of objects from the types of other objects. Data execution is more general than inheritance in this sense. In the latter case, subordinate structures are defined solely by aggregation (e.g., adding beds and carpets to rooms).

(b) Abstract operations play a central role on COO design and programming. They operate on objects as wholes – irrespective of the complexity of the class hierarchy corresponding to these objects – combining the effects of all semantically related functions. In our clean room example, the abstract operation clean might have make and vacuum as affiliates. Where the room in “clean (room)” is a bedroom, for example, clean would have the effect of cleaning generic parts of the room plus making the bed and vacuuming the carpet. Where the room is a kitchen, it would clean the generic parts plus the stove and sink.

As we saw above, abstract operations can be implemented by introducing the notion of affiliate functions. Abstract operations combine the effects of affiliated functions in an inheritance hierarchy. Overloaded
functions are just one example of affiliate functions. Implementation of these ideas in Flexsys is simplified because there is only one basic (node_ptr) type.

In effect, COO raises polymorphism to a higher level. Instead of just resolving overloading at runtime, cognitive (abstract) operations can incorporate any and all semantically related functions. They are viewed as working on the objects themselves – on any object where the operation makes sense semantically. In COO programming (as well as in design) one speaks of driving, cleaning or destroying a bicycle, car, train or airplane – not of the latter being drivable, cleanable and destroyable.

In this sense, the precedence of objects and functions in COO programming is reversed. Abstract operations act on “real world” objects. As noted above, abstract operations go beyond the realm of abstraction. The notion of affiliate functions makes them fully operational.

**Data Execution.** As noted above, data execution functions define relationships between objects (e.g., “done”). This important construct is directly supported in the HLD language. In C++ it is only enabled.

Thus, in COO programming we have:

```
[room_status] status (bed, carpet)
    [bed]: made
    [carpet]: dirty
  clean (room_status)
```

The same thing can be done in defining data structures. In HLD we have:

```
[room_object] construct (bed, carpet)
    [bed]
    [carpet]
  convert (room_object, bedroom_object)
```

where the latter statement refers to the conversion of (ordinary) rooms into bedrooms. In this case “convert” might simply add a bed and a carpet to the room (as is the norm in inheritance), or it might involve more esoteric operations.

In C++, or as an alternative in HLD, we have, respectively:

```
clean (status (bed, carpet))
```

and

```
convert (construct (bed, carpet), bedroom_object)
```
The former approach has the effect of replacing procedural complexity with structural complexity (cf. Scandura, 1981). This improves understandability in two ways: (a) it simplifies procedural statements and (b) it localizes date definitions.

**Abstract HLD Operations.** As noted above, abstract HLD operations go beyond simple polymorphism. In addition to allowing parameters to vary as to data structure, abstract operations, together with their affiliates perform the necessary behavior without specifying any additional procedural logic.

Consider the class hierarchy

```
[room]
  [bedroom]
    [bed]
      [carpet]
  [kitchen]
  ...
```

where bedroom and kitchen are categories of room, and bed and carpet are components of bedroom.

In HLD, without any additional work, we have:

```
clean (room)
```

irrespective of whether the room is a bedroom or a kitchen.

In C++, in addition to defining a room class hierarchy, we would have to write clean functions for both the bedroom and kitchen classes as well as for the room class. In the case of bedroom, for example, the clean function might call make and vacuum in the bedroom class plus clean in the room class. Another clean would be needed for the kitchen class. An alternative in C++ (or HLD) might be:

```
room.clean (room)  
bedroom.make (bed)  
bedroom.vacuum (carpet)
```

The disadvantage of the first (most common) approach is that: (a) we have more than one "clean to maintain and/or resolve when debugging (in addition to at compile or run time) and (b) the approach tends to camouflage the semantic equivalence of make and vacuum with respect to clean. The latter implementation provides more flexibility.
but it adds unnecessary procedural complexity and can easily lead to sloppy programs that are hard to maintain.

Further simplification derives from the fact that HLD functions and operations are typeless. All structures are explicitly represented hierarchically. This encourages the use of abstract objects (variables) in COO designs and programs. (The use of global variables is strongly discouraged with obvious advantages as regards system reliability, availability and maintainability.) Because parameters may be as abstract as necessary, one is never forced into using long parameter lists. Moreover, HLD operations are designed to facilitate modeling real-world behavior. Core HLD operations work on arbitrary data structures.

The HLD operation

`duplicate (structure_root, target_element)`

for example, works irrespective of the structure involved. It operates on arbitrary node structures (such as those shown in Figs. 1–4). Basic HLD operations, such as "add", ":=", etc. are implemented (in C++) so they generalize across data types (e.g., int, float, char). Indeed, all additions to any HLD library are wrapped in "nodes" to ensure that they are typeless and interoperable. Compound operations constructed from these operations also interoperate based solely on real-world semantics.

**Application of the Cognitive OO Programming Paradigm.** We have developed, tested and refined a significant number of complex COO programs written in the extensible HLD language. Many of these programs are of the higher order variety and are quite powerful. That is, they operate on and modify (or build) new programs. These higher order programs perform tasks ranging from constructing call hierarchies and reorganizing modules in large C or Ada programs to translating C or FORTRAN into "good" Ada (e.g., Scandura, 1994). Currently available COO programs, however, were built using more or less standard top-down methods. To date, full use of the cognitive OO paradigm has been limited. Indeed, systematic design debugging at successive levels of analysis was not possible until very recently, among other things because our Flexsys Simulator did not fully support the simulation of aliases (pointers).

By implementing operations on node_ptrs to "real world" objects, without reference to the internals as such, COO programming is a natural outgrowth of COO design and simplifies program representation. Not only do objects map directly into the real world but so do
abstract operations on those objects. People can describe systems during implementation as well as during design in the way they naturally think about what needs to be done—whether these abstractions refer to objects or to actions on those objects. Programs are described the way we speak and write—using verbs for actions and nouns for objects. Similar thinking, of course, also was a prime motivation for languages such as COBOL. At that time, however, modularization and the importance of structure were largely unknown, not to mention the possibility of encapsulating behavior with in data structures.

Characteristics of the COO paradigm are summarized in Table 1.

**REUSE OF LEGACY CODE**

The cognitive paradigm, especially when coupled with design debugging provides a natural foundation for the reuse of legacy code. Abstract models of given application domains often map naturally

<table>
<thead>
<tr>
<th>Table 1</th>
<th>General characteristics of the cognitive OO (COO) paradigm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Support for both “real world” objects and operations on those objects simplifies program representation. Systems are described like people think—whether abstractions refer to nouns (i.e., objects) or verbs (i.e., operations).</td>
</tr>
<tr>
<td>(2)</td>
<td>Hierarchical relationships between objects are a natural consequence of process refinement and <em>vice versa.</em></td>
</tr>
<tr>
<td>(3)</td>
<td>From a cognitive perspective, operations mapping real world input objects (onto output objects) correspond to conscious cognition. Functions defined between hierarchically related objects correspond in cognition to automatic (e.g., perceptual) processes—automatic actions not requiring conscious thought. Functions between objects give meaning to those objects—like “adjectives” modifying “nouns”.</td>
</tr>
<tr>
<td>(4)</td>
<td>There is no natural distinction between COO high and low level design. All distinctions are arbitrary and simply a matter of degree. The same is true of COO programming in the HLD language. The HLD language can be viewed as a preprocessor for C/C++—just as C++ was originally a preprocessor for C.</td>
</tr>
<tr>
<td>(5)</td>
<td>COO programs also can be implemented directly in C++ although the resulting syntax is often much more complicated. Generalizations of inheritance and evaluation, which arise naturally in COO programming, may be implemented in C++, albeit by resort to more complex constructions. Abstract COO operations may be implemented by aggregating overloaded C++ functions, by introducing typeless node_ptrs as parameters (i.e., by wrapping) or by affiliating functions in class hierarchies with the abstract operations.</td>
</tr>
<tr>
<td>(6)</td>
<td>The HLD language supports higher order programming in the COO paradigm. These higher order operations add “adverbs” to the real world modeling facilities.</td>
</tr>
</tbody>
</table>
onto components derived from legacy code in those domains. These components might be derived by simply reverse engineering and wrapping legacy code and/or they might be derived by first converting that legacy code into a common programming language like C++ or Ada (Scandura, 1994).

**Code Extraction and Semantic Wrapping.** To enable reuse, it is essential that legacy components be assessible by parameter passing—allowing direct, clean mappings between high level designs and to be reused components. These goals currently are accomplished by encapsulating potentially reusable routines in semantic wrappers. Semantic wrappers replace language-specific data types with semantically meaningful relationships between data. (Note: Depending on intended semantics, semantically wrapped functions in C++ classes may either be called as is or affiliated with functions in other, classes for external use.

The availability of needed components (once semantically wrapped) makes it possible to implement arbitrary real world models based on semantic considerations. Technical issues pertaining to specific data types and other code requirements (e.g., C/C++ or Ada syntax), as well as initialization and interface issues, are eliminated. Certain core (semantically wrapped) components provide a semantic “glue” enabling wrapped components from any source to work together seamlessly. Once wrapped, it makes no difference whether the components originally come from existing executables, purchased libraries or reusable legacy code.

The extraction and wrapping process is facilitated by a number of Flexsys' customizable re-engineering and conversion components which automatically gather information and/or otherwise improve legacy code.

**Semantics-Based (COO) Design.** Flexsys' visual (Flexform) Designer and Simulator environments make it possible to create, test and debug high level semantic-based models of arbitrarily complex, large scale software applications. Semantic core components provide “glue” making the (other) components referenced in these models interoperable. Referenced components may range in size from highly specialized routines to complete applications.

The above process can be used in developing any application. These applications can range from enterprise-wide systems (e.g., consisting of existing applications) to domain-specific application development systems for quickly constructing custom applications from wrapped components.
Applications built with this machinery are not tied to Flexsys' development environment. Once a new application runs as it should in the Simulator, full C/C++ source code can be generated automatically. Currently, Flexsys' "HLD Generator" and "Semantic Integrator" support the C/C++ language. Other languages could be supported via bindings.

Semantic Wrapping. Although details are beyond the scope of this article, some progress has been made in automating the semantic wrapping process. We have partially automated certain processes for standard C libraries, including reverse engineered (legacy) code. We also have designed solutions for C++ classes and for integrating legacy source code in other languages such as Ada. In addition to explicit methodologies, wrapping such things as middleware (e.g., for distributed computing), OLE and CORBA compliant objects, Motif widgets and integration with event driven code generated by modern GUI builders is a priority.

Application to a Large C System. The cognitive paradigm has been successfully applied in re-engineering a large C system. Specifically, cognitive methodology and supporting re-engineering technology were used to completely redesign "top level" code in the PRODOC software engineering system. In the process, a large number of C components were extracted (from a system of 500 K LOC), semantically wrapped and made interoperable. Originally, none of this work directly effected functionality of the system. The system did exactly what it had done before. From a maintenance (and enhancement potential) standpoint, however, the differences were dramatic.

Before redesign, the highest level compilation unit in the system consisted of 72 C Flexform modules. After the redesign, all of the design logic is contained in only five HLD Flexforms. These Flexforms call about 65 semantically wrapped HLD Flexforms comprising the bulk of the lower level code. The latter are referenced resources but otherwise may be viewed as "black boxes". As a result of the top level redesign, the five design Flexforms are the only ones which need to be changed. Moreover, all of the semantically wrapped components are interoperable, not only with each other but with every other semantically wrapped component. As a result, we have observed a dramatic reduction in the effort required to maintain, enhance and otherwise modify overall system logic. In this regard, length alone was reduced by a factor of 12 or more. (In terms of maintainability, one can make a case for complexity increasing by the square of the length – i.e., a factor of 144.)
CONCLUSIONS

The COO paradigm represents a potentially important advance in modeling and building software, offering a more complete representation of reality than does the OO paradigm. Among other things, cognitive technology helps to simplify and/or eliminate problems which have plagued the OO paradigm from its inceptions. Cognitive (COO) design and programming allows more natural "real world" modeling and makes it easier to test and debug systems.

On the pragmatic side, COO programming can be implemented in C/C++. The generation of compilable C source code from COO designs is currently available and fully automatic. In this paradigm, the only significant change required is the increased design discipline it supports. COO design discipline minimizes integration problems associated with traditional programming and design methodologies.

Nonetheless, the above represents only a beginning. COO design derives from a long history of research in structural (cognitive task) analysis and has a relatively sound foundation at this point. COO principles also have been used successfully in redesigning and re-engineering a large (500 K LOC) C system. Though the results to date have been relatively clear cut, full implications for developing and maintaining large software systems are still to be established.

References


Scandura, J.M. Problem solving in schools and beyond: Transitions from the naive to the neophyte to the master. *Educational Psychologist*, 1981, 16, 139–150.


