Journal of Systems Integration, 8, 255–285 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

The Algebraic Framework for Object-Oriented Systems

D. H. H. YOON dhyoon@umdsun2.umd.umich.edu
Department of Computer & Information Science, University of Michigan–Dearborn, Dearborn, MI 48128

Abstract. Category Theory is introduced as the mathematical model for object-oriented systems which are distributed, heterogeneous, real-time, embedded, and open-ended. Each object can be represented as an algebra. A collection of algebras with morphisms form a category if they satisfy some conditions. After a brief introduction of categorical concepts which are needed to formulate the framework for object-oriented systems, they are explicated in terms of objects. Then some system design methodologies such as SADT, JSD, MASCOT 3, OOD, HOOD, MOON, ADM 3, and Petri nets are examined in the categorical framework and classified into four groups: functional, process-based, object-oriented, and net-based. Combining theoretical and practical results, the interactive system design tool OBJ-NET is briefly introduced.

Keywords: Algebras, category, object, composition, union, system, design language, diagram

1. Introduction

Objects have emerged as a new computing paradigm generalizing the notions of a Turing Machine and an algorithm [7]. For this reason they have been employed in all branches of Computer Science, hardware design, software engineering, database, information systems, networks, computer graphics, system design, etc. Due to the wide variety of applications, no two people agree on the notion of an object. It is imperative to establish the theoretical foundation of Object-Oriented Computing at least for two reasons: First, to bring order out of the chaotic activities in object-oriented technology. Second, to shed light on future research directions in the area.

Most literature on object-oriented technology deals with methodologies rather than its essence. Consequently it results in a methodology jungle. Application oriented users are often left with the impression that objects are something new and are not related to traditional computing paradigms such as Turing Machines and algorithms which are implemented using primitive data types (PDTs) and abstract data types (ADTs). On the contrary, objects have evolved from all of the above and generalize them. In order to establish the continuity between traditional and object-oriented computing, their fundamental computing models are examined in the algebraic framework in which objects and the similar entities such as PDT, ADT, and module are represented as algebras [1–6]. In practice, objects rarely exit in isolation. Instead, they co-exit with other objects, communicate with one another, and form a community or society of objects. The notion of a category is introduced to represent a community of objects. Once categorical concepts are introduced, the graphical representation technique of the objects in a category is readily available. This is crucial because diagrams have been employed as a design tool for many years in Engineering, yet they have not been treated systematically. Category Theory allows one to integrate

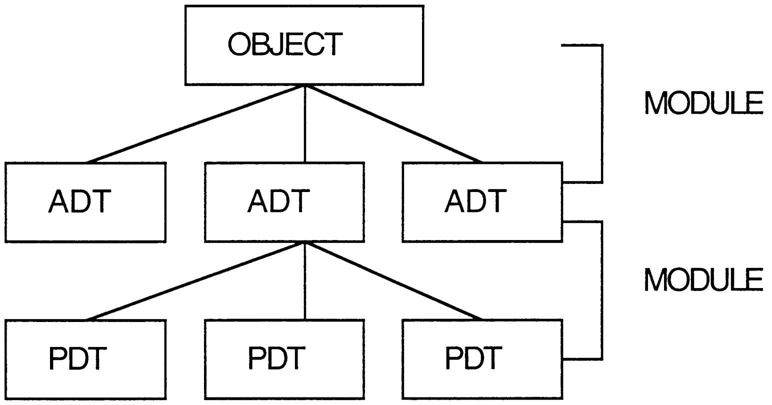


Figure 1. The hierarchy of computational models.

algebraic specification, diagramming, and programming into one entity. Most system design methodologies employ design languages and graphical notations. In this paper, major system design methodologies such as SADT, JSD, MASCOT 3, MOON, HOOD, ADM and Petri Nets are examined in the category-theoretical context concentrating on their views of a system, design languages and graphical notations [3]. It turns out that the notion of a system evolves along with the changing computer technology, and design languages and graphical notations become more sophisticated as time goes on.

In the next section, an algebra is introduced as the canonical form for representing PDTs, ADTs, modules, and objects. This is the common concept which underlies both traditional and object-oriented computing. Hence, it is not hard to represent Turing Machines, algorithms and objects in the algebraic form [7].

An *Open System* is defined as a net of objects. Then, some important system design methodologies are examined in the categorical context. It turns out that all of the above design methodologies use textual and graphical representation schemes, which are equivalent to algebras and diagrams, respectively, in the categorical framework. Finally, OBJ-NET, an interactive system design tool, is introduced, which utilizes both theoretical and practical findings on objects [3, 4].

2. Algebras

Through a series of papers [1–3], the author has established an object as *an abstraction or computational model of a real world entity*. This definition of an object is broad enough to include the various views of an object such as Alan Kay's stating that everything is an object [8], an object as a model of a real world entity shared by many [11–12], Hewitt's actors [13], Shoham's agents [14], and Krief's prototypes [15].

Ever since objects were introduced in SIMULA [16] in 1967 the idea of an object has

```
OBJ   stack
      SORTS       item, list;
      OPNS
            push : list $*$ item $\longrightarrow$ list;
            pop : list $\longrightarrow$ item;
      EQNS
            for all i $\in$ item, l $\in$ list,
            push (l,i) $\longrightarrow$ l;
            pop (l) $\longrightarrow$ i;
ENDOBJ.
```

Figure 2. The algebraic specification of object STACK.

evolved continuously. As one can easily see from the above variants, new names have been introduced as new features were added to the original objects. For instance, an actor is an object with communication capability added and an agent is an actor with intelligence added. There are almost infinitely many terminologies introduced in conjunction with objects. This is very confusing. In order to avoid this type of confusion and to put object-oriented technology on the firm theoretical foundation, the author has been advocating the algebraic framework for the technology.

As Fig. 1 illustrates, an object is based on one or more abstract data types (ADTs) and an ADT consists of a few primitive dat types (PDTs). Modules play the major role in specifying an ADT in terms of PDT's and an object in terms of ADTs. Procedures and functions in programming languages are typical examples of modules. In general, a module refers to "a separable component, frequently one that is interchangeable with others, for assembly into units of differing size, complexity, or function" [16]. There is a common structure among these entities, PDT, ADT, module, and object, which can be expressed as an algebra.

To be specific, consider the following representation of PDTs: Boolean $= \langle B, \{\text{AND}, \text{OR}, \text{NOT}\}\rangle$, Char $= \langle C, \{\text{ord}, \text{chr}, \text{pred}, \text{succ}\}\rangle$, Integer $= \langle I, \{+, -, *, /\}\rangle$, Real $= \langle R, \{+, -, *, /\}\rangle$, where B, C, I, and R represent boolean, character, integer, and real variables, respectively. Similarly, ADTs can be represented in the same form: stack $= \langle \text{list}, \{\text{push}, \text{pop}\}\rangle$ and queue $= \langle \text{list}, \{\text{insert}, \text{delete}\}\rangle$. One can easily see the common structure underlying both PDTs and ADTs: an entity is represented as an encapsulation of a set of data and operations defined on the set, i.e. entity $= \langle \text{data}, \text{operations}\rangle$.

An object can be represented in a similar form. The specification of the stack object in Fig. 2 uses the UMIST OBJ notation [18] and introduces terminologies that we need to define an algebra. In the above specification, OBJ, SORTS, OPNS, EQNS, and ENDOBJ are reserved words. The object stack is specified in the algebraic form which consists of SORTS and OPERATIONS. A sort is equivalent to a data type in programming languages, and the operation section, denoted by OPNS, is equivalent to the declaration part of a programming module in which all the variables, functions, and procedures are declared.

The OPNS section is equivalent to a *signature* of an algebra. The EQNS section is equivalent to the body of the module in which all the functions and procedures are defined.

Using the above intuitive meanings of a sort and a signature, an algebra is formally defined as follows:

Definition 1. If Σ is a signature, a Σ-algebra is a pair $\langle S, \Sigma_S\rangle$ where

i. S is the set of sorts,

ii. Σ_S is the set of operations $\{f\}$ such that if arity $(f) = n$, then $f_S\colon S_1 \times S_2 \times \cdots \times S_n \to Sk$, where $1 \leq k \leq n$.

One can easily see that in the stack specification (Fig. 2), $S =$ sorts and $\Sigma_S =$ EQNS. An algebra is a mathematical mechanism to encapsulate data and operations and forms the basis of both procedural and object-oriented computing. Furthermore, it implies the existence of hardware devices which provide the space for data or perform the operations. To be specific, a set of data requires a block of memory in RAM or ROM and the operations are carried out by CPU or an I/O processor. Although in this paper the emphasis is placed on the software aspects of object-oriented technology, it is beneficial to keep in mind that there are hardware devices associated with software components.

A set of objects form a category if they satisfy some conditions. In the next section, a few categorical concepts, which are relevant to object-oriented technology, are introduced.

3. Some Categorical Concepts

Category Theory enjoys a wide range of applications in Computer Science including the design of programming languages, the development of concurrent models, type theory, polymorphisms, automata theory, database, etc. The advantage of Category Theory is that it can be easily applied to a specific domain without introducing the entire body of the theory. In this section only a few very fundamental concepts of Category Theory, which are relevant to Object-Oriented System Theory, are introduced. The reader who is interested in this line of research is referred to Pierce [19] or Barr and Wells' book [20].

Definition 2. A category **C** comprises:

1. A collection of objects (algebras);
2. A collection of arrows (morphisms);
3. Operations assigning to each arrow f an object dom f, its domain, and an object cod f, its codomain (we write $f\colon A \longrightarrow B$ to show that dom $f = A$ and cod $f = B$).
4. If arrows $f\colon A \to B$, $g\colon B \to C$, and $h\colon C \to D$ exist, then $h * (g * f) = (h * g) * f$.
5. For each object A, an identity arrow $\mathrm{id}_A\colon A \to A$ satisfies the following law: For any arrow $f\colon A \to B$, $\mathrm{id}_B * f = f$ and $f * \mathrm{id}\, A = f$.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"'] & Z \arrow[d, "g'"] \\
W \arrow[r, "f'"'] & Y
\end{tikzcd}$$

Figure 3. A commutative diagram.

Since an object can be represented as an algebra, a category can be viewed as a set of algebras along with morphisms defined on them. The algebraic representation of a category becomes complicated very quickly and overwhelming. For this reason the graphical representation technique, diagraming, has been used extensively in Category Theory.

Definition 3. A diagram **D** in a category **C** is a collection of vertices and directed edges, labeled with objects and arrows of **C**.

For simplicity, a diagram and a category will be denoted as follows: Diagram $\mathbf{D} = \langle V, E \rangle$ where V is a set of vertices and E is a set of edges and Category $\mathbf{C} = \langle O, A \rangle$ where O is a set of objects and A is a set of arrows.

Having defined a diagram, we explicate a category in terms of its diagram. Consider a category C which consists of objects $\{X, Y, Z, W\}$ and arrows $\{f, f', g, g'\}$, i.e. $C = \langle O, A \rangle$, where $O = \{X, Y, Z, W\}$ and $A = \{f, f', g, g'\}$. Then its diagram D might look like the one in Fig. 3. A Diagram **D** in a category **C** is said to *commute* if, for a pair of vertices X and Y with edges $f\colon X \to Z$ and $g'\colon Z \to Y$, there exist edges $g\colon X \to W$ and $f'\colon W \to Y$ such that $g' * f = f' * g$, for some $Z, W \in O$ and $f, g, f'g'A$.

Example 1. Let us consider a system consisting of four heterogeneous objects which are connected by a net. The system may be represented as a category: $\mathbf{C} = \langle O, A \rangle$ where $O = \{\mathrm{obj}_1, \mathrm{obj}_2, \mathrm{obj}_3, \mathrm{obj}_4\}$ and $A = \{f\colon \mathrm{obj}_1 \to \mathrm{obj}_2, g\colon \mathrm{obj}_2 \to \mathrm{obj}_3, h\colon \mathrm{obj}_3 \to \mathrm{obj}_4, g * f\colon \mathrm{obj}_1 \to \mathrm{obj}_3, h * g\colon \mathrm{obj}_2 \to \mathrm{obj}_4, h * (g * f)\colon \mathrm{obj}_1 \to \mathrm{obj}_4\}$. Properties (1), (2), (3) and (5) of Definition 2 are obvious. Only property (4) is explicated in some detail.

Since each object in the system is assumed to be heterogeneous, each object has its own architecture and language. A typical example of this type of system is a manufacturing system consisting of milling machines, lathes, transport systems, etc. [2]. Each piece of equipment is represented as an object. In order for them to communicate, physical links

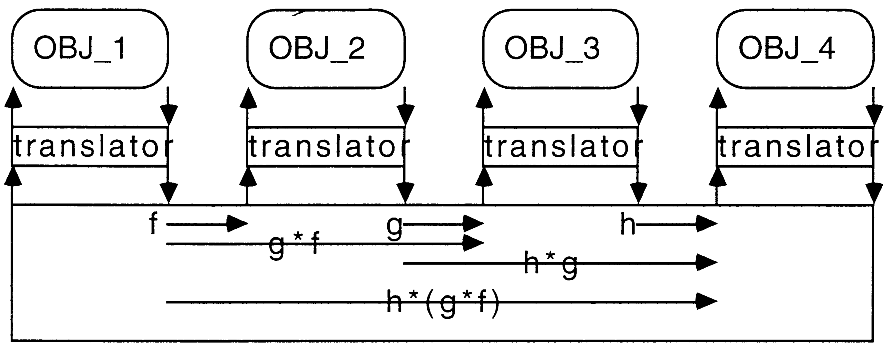


Figure 4. A system as a category.

$$\begin{tikzcd} & C \arrow[dl, "f"'] \arrow[d, "<f,g>"] \arrow[dr, "g"] & \\ A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B \end{tikzcd}$$

Figure 5. A product of two objects.

and protocols are required. Because each object uses its own language, there is a need for a translator which translates local languages to the universal language and vice versa.

In order to illustrate property (4) of definition 2, arrows f: $\text{obj}_1 \rightarrow \text{obj}_2$, g: $\text{obj}_2 \rightarrow \text{obj}_3$, and h: $\text{obj}_3 \rightarrow \text{obj}_4$ represent the communication channels among the corresponding objects (Fig. 4). Arrows $g * f$: $\text{obj}_1 \rightarrow \text{obj}_3$, $h * g$: $\text{obj}_2 \rightarrow \text{obj}_4$, and $h * (g * f)$: $\text{obj}_1 \rightarrow \text{obj}_4$ indicate composite channels among the corresponding objects. One way of interpreting $h * (g * f)$ is that obj_1 sends a message to obj_3 first and obj_3 sends it to obj_4. On the other hand $(h * g) * f$ indicates that obj_1 sends a message to obj_2 first, and then obj_2 sends it to obj_4. Hence obj_1 can send a message to obj_4 through different channels. Both channels deliver the same message from obj_1 to obj_4. Fig. 4 is the diagram D of the category C.

A category needs to be built up from primitive objects. For this, two operations, *product* and *coproduct* are defined. In order to define them, some notations are needed. Let A and B be sets. Then the cartesian product of two sets A and B is $A \times B = \{(a, b) \mid a \in A$ and $b \in B\}$. When the cartesian product of two sets A and B is formed, the corresponding projection functions π_1: $A \times B \rightarrow A$ and π_2: $A \times B \rightarrow B$ are implicitly defined.

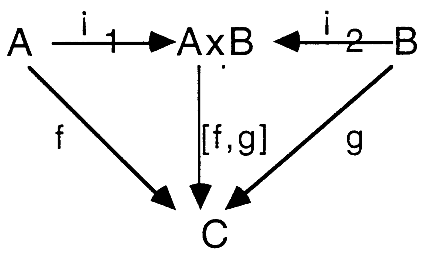


Figure 6. A coproduct of two objects.

Unlike the cartesian product of two sets, a product of two objects requires the cartesian product of the data sets along with the product of operations. For this, assume that for some set C, there are two operations $f\colon C \to A$ and $g\colon C \to B$. Then we can form a product operation $\langle f, g\rangle\colon C \to A \times B$ such that $\langle f, g\rangle(x) = \langle f(x), g(x)\rangle$. The operations f and g can be recovered from $\langle f, g\rangle$ by setting $f = \pi_1 * \langle f, g\rangle$ and $g = \pi_2 * \langle f, g\rangle$.

Definition 4. A *product* of two objects A and B is an object $A \times B$, together with two projection arrows $\pi_1\colon A \times B \to A$ and $\pi_2\colon A \times B \to B$, such that for any object O and pair of arrows $f\colon C \to A$ and $g\colon C \to B$, there is exactly one mediating arrow $\langle f, g\rangle\colon C \to A \times B$ making the diagram commute, i.e., $\pi_1 * \langle f, g\rangle = f$ and $\pi_2 * \langle f, g\rangle = g$.

A product of two objects is similar to the cartesian product of two sets. However, since each object has morphisms associated with it, a product of two objects requires the cartesian product of two sets of sorts and the product of the morphisms associated with the two objects. This concept is illustrated in terms of the bivariate normal distribution function.

Example 2. Let X and Y be independent random variables on the real line with the standard normal distribution, i.e., $X \sim N(0, 1)$ and $Y \sim N(0, 1)$. Then objects $A = \langle X, f_x()\rangle$ and $B = \langle Y, f_y()\rangle$, where $f(u) = 1/(2\pi)^{1/2} \exp(-u^2/2)$, where $-\infty \leq u \leq \infty$. Then the product $C = A \times B = \{X \times Y, \langle f_x(), f_y()\rangle\}$, where $\langle f_x, f_y\rangle = f_x(u) * f_y(v) = 1/2\pi \exp -(u^2 + v^2)/2$ and $-\infty < u, v < \infty$.

The dual notion, *co-product,* corresponding to set-theoretic disjoint union is defined as follows:

Definition 5. A *co-product* of two objects A and B is an object $A + B$, together with two injection arrows $\iota_1\colon A \to A + B$ and $\iota_2\colon B \to A + B$ such that for any object C and pair of arrows $f\colon A \to C$ and $g\colon B \to C$ there is exactly one arrow $[f, g]\colon A + B \to C$ making the following diagram commutes:

The co-product of two objects is very similar to the disjoint union of two sets except that it involves combining morphisms. The following example from Computer Network will illustrate the concept.

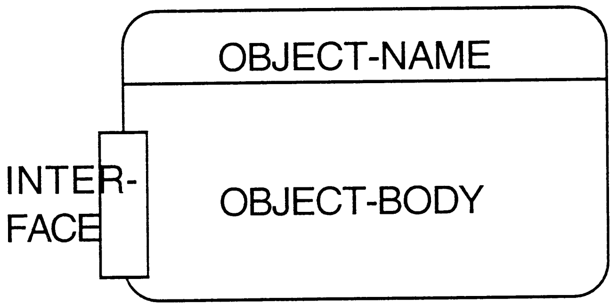


Figure 7. A graphical representation of an object.

Example 3. Group Communication refers to the communication mechanism in which a member of a group communicates with the rest of the members by broadcasting a message to everyone in the group. Other members, upon receiving the message, respond to the message only if it is addressed to them. Let $A = \langle G_1, f\rangle$ be an object where $G_1 = \text{group1} = \{m_{1i}, i = 1\ldots a\}$ and $f =$ the broadcasting mechanism defined on G_1, and $B = \langle G_2, g\rangle$ be another object with $G_2 = \text{group2} = \{m_{2j}, j = 1\ldots b\}$ and $g =$ the broadcasting mechanism defined on G_2. Then the composite object $C = A + B = \langle G_3, [f, g]\rangle$, where $G_1 \cup G_2 = G_3$ and $[f, g] =$ the broadcasting mechanism defined on G_3. The new group G_3 includes groups G_1 and G_2 and the broadcasting mechansim $[f, g]$ includes the mechanisms f and g defined on G_1 and G_2, respectively [10].

In this section, a category, a diagram, and product and co-product are introduced. They are sufficient to formulate the framework for object-oriented system theory which permits to integrate diverse activities in object-oriented technology. In the next section, the above categorical concepts are interpreted in terms of objects and their block-diagrams.

4. The World of Objects

Putting together what has been presented so far, we introduce three representation techniques of an object: algebraic, graphical, and modular [1–3]. The algebraic representation of an object leads to the area called algebraic specification [6], the graphical representation encompasses various diagraming techniques, and the modular representation includes object-oriented programming languages. A great deal of work has been done on algebraic specification and object-oriented languages. Although diagrams have been employed widely in practice, they have never been treated formally. Category Theory allows one to unify the three representation techniques into one.

As Fig. 2 illustrates, the algebraic specification of an object consists of the object name, sorts, operations (OPNS), equations (EQNS), i.e., $\text{OBJ} = \langle \text{SORTS}, \text{OPNS}, \text{EQNS}\rangle$. An object can be represented as a black box and viewed as a miniature system. Fig. 7 employs the HOOD notation to represent an object pictorially.

4.1. Operations on Objects

In order for objects to be useful, they need to be related to other objects. Two operations, *product* and *co-product,* introduced in the previous section provide the two fundamental ways of building larger objects from small ones. Instead of using the generic category theoretical terminologies, two new, but equivalent terminologies, *composition* and *union*, which are more familiar to computer professionals, are introduced.

4.1.1. Composition

The *composition* of two objects is equivalent to the *product* introduced in the previous section. It allows one to come up with a larger object by combining two smaller ones. To be specific, let us consider the following examples:

```
OBJ O1                  OBJ O2
   SORTS_1 ---;               SORTS_2 ---;
   OPNS_1  ---;               OPNS_2 ----;
      -----;                      ----;
      -----;                      ----;
   EQNS_1 ----;               EQNS_2 ----;
      ----;                     ----;
ENDO1.                  ENDO2.

OBJ O3
   USES O1, O2;
   SORTS_3 -----;
   OPNS_3 ----;
      -----;
   EQNS_3----;
      ----;
ENDO3.
```

OBJ O3 is obtained by composing OBJs O1 and O2. A Composition of O1 and O2 into O3 may be interpreted as a product of two objects as follows. Notice O1 $= \langle$SORTS_1, OPNS_1, EQNS_1$\rangle$, O2 $= \langle$SORTS_2, OPNS_2, EQNS_2$\rangle$ and O3 $= \langle$SORTS_3, OPNS_3, EQNS_3$\rangle$. SORTS_1, SORTS_2 and SORTS_3 are simply sets of data types, both primitive and abstract, and OPNS_1, OPNS_2 and OPNS_3 are sets of operations. Now, SORTS_3 $\supset$ SORTS_1 $\times$ SORTS_2, where the '$\times$' indicates the cartesian product of two sets of data types. OPNS_3 $\supset$ OPNS_1 $\times$ OPNS_2 where OPNS_1 $= \pi_1$ (OPNS_1 $\times$ OPNS_2) and OPNS_2 $= \pi_2$ (OPNS_1 $\times$ OPNS_2).

The following example in C++ further illustrates this concept.

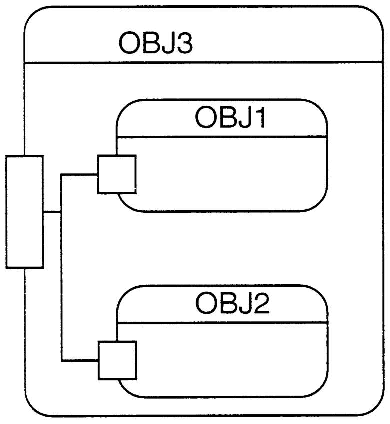


Figure 8. The composition of two objects.

Example 4. The following C++ code declares object PC as a composition of sub-objects DISK, KEYBOARD, and SCREEN.

```
class DISK { int rpm;
         float price;
      public
         DISK ( int R, float P); rpm (R), price (P){}:
        }

class KEYBOARD { int numkeys;
         float price;
      public
         KEYBOARD (int N, float P);
         numkeys (N), price (P){};
        }
class SCREEN {char colorflg;
         float price;
      public
           SCREEN (int C, float P);
           colorflg (C), Price (P) {}:
        }
class PC { long RAM;
         long ROM;
         char * cpu_type'
         float price;
         DISK storage;       // three sub-objects
```

```
            KEYBOARD kb;
            SCREEN crt;
          public
                PC (long, long, char *, float, int, float, int,
                    float, int, float);
               ~PC;
          }
```

The above code is the skeleton of the declaration of class PC as a composition of sub-objects DISK, KEYBOARD, and SCREEN. For the complete one, the reader should refer to [9].

Another very useful operation on objects is *union* which enable use to link two or more objects. Unlike *composition*, this operation requires an entity encapsulating hardware, software, and interface components. In the example (Fig. 4) in which a net of objects is illustrated as a category, the reader has seen that in order to link two objects a network bus and translators were employed. This is true at any level of systems. When system components are linked, it requres a system bus and sending and receiving mechanisms should be worked out at the software level. In the next section, the concept of a union is further explained.

4.1.2. Union

Union is equivalent to the co-product defined in section 3 and enables one to connect two or more independent objects in terms of a communication link. The client-server model is a typical example of this operation. In this example, the client and the server are linked together in terms of a net or channel. Hence, LAN, MAN, and WAN are special kind of nets. The union of two objects will be denoted by $O3 = O1 + O2$. The communication between the client and the server is achieved through remote procedure calls or a group communication mechanism.

When dealing with processes which are also objects, the communication between them is achieved through inter-process communication mechanism such as pipes, FIFO's, message queues, semaphores, and shared memories. When a process is broken down into modules, modules communicate with each other through parameter passing.

A union (notice that it is not 'the' union) of two objects involves the disjoint union of two sets of sorts and combining the corresponding operations. Using the same notation as in the previous section, SORTS_3 $\supset$ SORTS_1 $\cup$ SORTS_2, where the $\cup$ indicates the disjoint union of two sets. OPNS_3 $\supset$ OPNS_1 $\cup$ OPNS_2 where OPNS_1 $= \pi_1$ OPNS_3 and OPNS_2 $= \pi_2$ OPNS_3. Example 5 below along with example 3 should illustrate the notion of a union pretty clear.

Example 5. As an example of a union of two objects, the client-server model is discussed. In Fig. 9, $O1 =$ client and $O2 =$ server. In order to connect heterogeneous objects, the physical link (or network bus) and translators (commonly known as stubs) are required. $O3$ is the new system obtained by linking $O1$ and $O2$ through other entities such as the physical

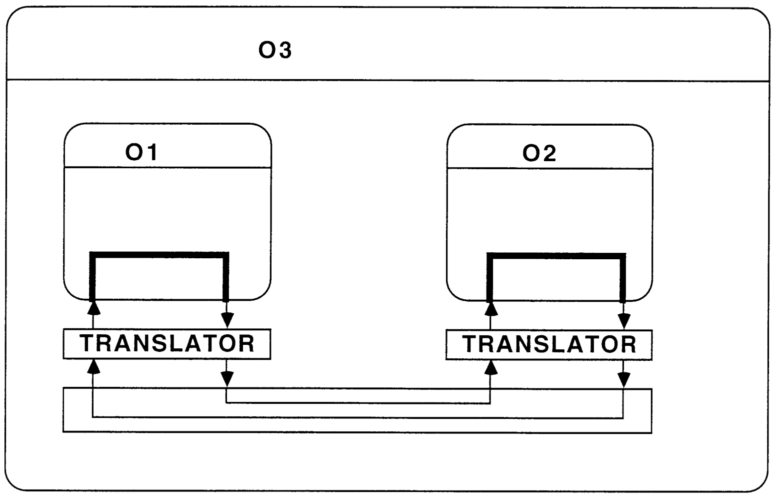


Figure 9. Object $O3$ as a Union of Objects $O1$ and $O2$.

link, translators, and network servers. The entire operations available in the network are the collection of operations provided by $O1$ and $O2$. That is, $\mathrm{OPNS}_3 = \mathrm{OPNS}_1 \cup \mathrm{OPNS}_2$.

The following code illustrates how the client requests the server for service [10]. However, in order to avoid the considerable overhead of the connection-oriented protocols, the following code is based on connectionless *request/reply* protocols. The client sends a request message to the server asking for some service. The server does the work and returns the data requested or an error code indicating why the work could not be performed.

```
 //
 // Both the client and the server should include the following
 // header file.
 //
#define MAX_PATH 255 // maximum length of a file name
#define BUF_SIZE 1024 // how much data to transfer at once
#define FILE_SERVER 243 // file server's network address
 //
 // Definition of the allowed operations
 //
#define CREATE  1 // create a new file
#define READ  2 // read a piece of a file and return it
#define WRITE   3 // write a piece of a file
#define DELETE  4 // delete an existing file
```

```
 //
 // Error codes
 //
#define OK  0 // operation performed correctly
#define E_BAD_OPCODE 1 // unknown operation requested
#define E_BAD_PARAM 2 // error in a parameter
#define E_IO  3 // i/o error
 //
 // Definition of the message format
 //
struct message {
 long source;  // sender's identity
 long dest;  // receiver's identity
 long opcode;  // which operation: CREATE, READ, ETC
 long count;   // how many bytes to transfer
 long offset;  // where in  a file to start reading
              // or writing
 long extra1;  // extra field
 long extra2;  // extra field
 long result;  // result of the operation reported here
 char name[MAX_PATH]; // name of the file being operated on
 char data[BUF_SIZE]; // data to be read or written
 };
 //
 // The Declaration of the Server;
 //

#include <header.h>

void main (void)
 {
 struct message m1, m2   // incoming and outgoing messages
 int r;     // result code
 while (1) {    // server runs forever
 receive (SILE_SERVER, &m1)  // block waiting for a message
 switch (m1.opcode) {   // dispatch on type of request
 case CREATE: r=do_create (&m1, &m2); break;
 case READ: r=do_read(&m1,&m2); break;
 case WRITE:  r=do_write (&m1,&m2); break;
 case DELETE:  r=do_delete (&m1,&m2); break;
 default: r=E_BAD_OPCODE;
 }
 m2.result = 4;   // return result to client
 send (m1.source, &m2);  // send reply
}
```

```
//
// The declaration of the Client
//


#include <header.h>

int copy (char *src, char *dst)  // procedure to copy file
                                 // using the server
 {
 struct message m1;    //message buffer
 long position;    // message buffer
 long client = 110;   // client’s address
 initialize ();    // prepare for execution
 position = 0;
 do {
 // get a block of data from a source file
 m1.opcode = READ;   // operation is a read
 m1.offset = position;  // current position in the file
 m1.count = BUF_SIZE;  // how may bytes to read
 strcpy (&m1.name, src);  // copy name of file to be read to
                          // message
 send (FILE_SERVER, &m1); // send the message to the file
                          // server
 receive (client, &m1);  // block waiting for the reply
 // Write the data just received to the destination file
 m1.opcode = WRITE;   // operation is a write
 m1.offset = position;  // current position in the file
 m1.count = m1.result;  // how many bytes to write
 strcpy (&m1.name, dst);  // copy name of file to be written
                          // to buf
 send (FILE_SERVER, &m1); // send the message to the file
                          // server
 receive (client, &m1);  // block waiting for the reply
 position += m1.result;  // m1.result is number of bytes written
     } while (m1.result > 0);  // iterarte until done
 return (m1.result >= 0 ? OK: m1.result);  // return OK or
                                           // error code
}
```

In this particular example it is assumed that there are only one client and one server. However. this can be generalized to a situation in which there are multiple servers and clients. Then the network server maintains a list of member objects along with operations they perform.

4.2. An Open System

Applying repeatedly the two operations, *composition* and *union*, one can easily build up an open system and define it as follows:

Definition 6. An open system is a net of objects, interacting with its environment.

The net in the above definition is a realization of the union operation on objects and connects two or more objects. The above definition generalizes various views on a system:

i. a system as a black box [21],

ii. A system as a society of communicating sequential processes (CSP) [22–23],

iii. A system as a collection of communicating experts [24],

iv. A system as a collection of modules [25–26],

v. A system as an interconnection of objects [27].

When an object is employed as a component of a system, it is considered as a computing agent which is autonomous and capable of communicating with other agents. An object is autonomous in the sense that it possesses its own hardware, software, and interfacing components. The definition above essentially states that a system is a collection of autonomous objects. Hence, the system is distributed in that each object has its own hardware, software, and interface and also heterogeneous in that the architectures of the objects in the system are distinct. Once a system is developed, it is often incorporated into a larger system. For example, an engine controller is incorporated into an automobile. This type of system is called embedded. Our definition of a system is broad enough to include the following systems:

i. distributed

ii. heterogeneous

iii. real-time

iv. embedded

v. open-ended.

A system is open-ended if some components can be added or deleted without damaging the integrity of the system and real-time if it requires that computation be correct and completed within a time limit.

A typical example of an open system is a Computer Integrated Manufacturing (CIM) system which consists of robots, milling machines, lathes, a transport system, computers, etc. [2].

5. System Design

Ever since computers were introduced to the mankind, they have been used as the controllers of various systems. For example, digital watches, microwave ovens, dishwashers, automobile engine controllers, automatic pilot systems, CIMs, etc are controlled by computers or microchips. Simple systems can be modeled as finite-state machines. However, for large and complex systems the finite-state model alone is not adequate. As systems become more complex, there have been infinitely many design methodologies introduced and they have formed the "methodology jungle." We propose the uniform system theory based on objects and nets.

Our object-oriented system theory incorporates most existing system design methodologies as special cases. Two operations, composition and union, allow objects to be composed (or decomposed) and linked (or unlinked). These operations along with their inverses allow one to specify the structure of complex computer controlled systems which are special cases of object-oriented systems and bring order out of the chaotic design activities.

Elaborating on the definition of a system stating that it is a net of objects interacting with its environment (definition 6), Fig. 10 demonstrates the hierarchy of a system along with the primary object at each level. A system consists of subsystems, a subsystem consists of hardware, software, and interface, a software system comprises of objects, an object consists of processes, a process can be viewed as a special kind of an abstract data type (ADT) defined in terms of a number of PDTs.

There are numerous system design methodologies proposed, most of which deal with designing software systems and can be classified in terms of how they view a system as follows:

1. Functional approach: SADT
2. Process based approach: JSD, MASCOT 3
3. Object-Oriented approach: MOON, HOOD, ADM
4. Net based approach: Petri net

Each of the above approaches has its own view of a system, a design language which is equivalent to an algebra, and a graphical representation which is equivalent to a diagram. In the next section, they are examined in terms of their views of a system, design languages, and graphical notations.

5.1. Functional Approach

Traditionally, a system has been viewed as a black-box which transforms input to output. In the functional approach, the black-box is considered as a function, and what a system does or the function of a system is the major concern of a designer.

This approach has a deep root in the functionality of CPU whose primary task was considered to perform mathematical functions by executing arithmetic and logical operations.

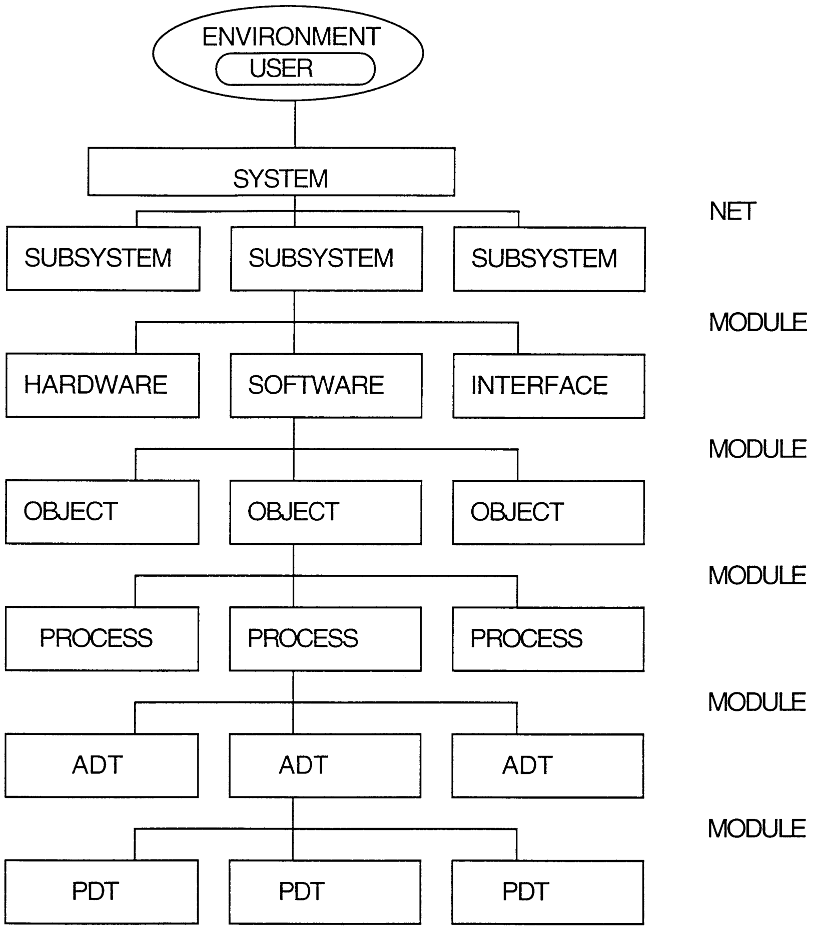


Figure 10. The hierarchy of system components.

Coupled with the concept of a subroutine, large functions are decomposed into smaller ones.

The purpose of a system is believed to do something or to perform some function. The essential content of a system specification is a statement of system function using both a design language and a graphical notation. The system function normally is not simple and needs to be decomposed into smaller ones. The system function is organized as a hierarchy of subfunctions, and design involves elaborating this hierarchy in the top-down fashion. The functional approach, in a way, is based on the external view of a system rather than its internal organization.

5.1.1. SADT [28]

Among many system design methodologies of this approach, Structured Analysis and Design technique (SADT) is best known. It was developed in the late 1960's and used successfully by the U.S. Air Force for the development of Integrated Computer Aided Manufacturing (ICAM) in the mid 1970's and the early 1980's.

In SADT, a system consists of interacting components with relationships among them, i.e., system = ⟨components, relationships⟩. Although the system is viewed as a collection of components along with their relationships, the activities or the functions of each component are of supreme importance. SADT employs both natural and graphical languages to describe the activities of the system. In particular, the SADT diagraming scheme uses boxes and arrows, i.e., diagram = ⟨boxes, arrows⟩, where a box may represent an activity of the system under development, a function, a process, or a real world entity, whereas an arrow may represent an interface or interconnection between two boxes, a relationship between two entities, input, or output. A box may be decomposed into many boxes and may have more than one arrows. Arrows may take different shapes, branch, or join. For further properties of the SADT boxes and arrows, Marca and Gowan's book [24] is highly recommended.

SADT combines both diagrams and textual representations. The skeleton of a system can be described in terms of a set of diagrams. However, details are left out in diagrams. Natural languages like English are needed for communication between various groups of people, for instance, designers, programmers, maintainers, and end-users.

SADT is one of the most successful system design methodologies and is employed even in the 1990's. The significance of SADT lies in the following:

(i) It lays down the groundwork for the development of computer controlled systems by defining a system as a collection of components along with their relationships.

(ii) It employs a design language and a graphical notation which are the two major tools of any system design methodology.

(iii) It planted seeds for the development of subsequent design approaches.

5.2. Process Based Approach

This approach has evolved out of the functional approach. In the functional approach, a system is viewed as a collection of interacting components along with the relationships among them. However, the emphasis is placed on the activities or functions of each component. In the process based approach, the components themselves are emphasized. Of course, each component performs one or more functions. In a way a process is a miniature system. In this approach, a system is viewed as a collection of processes, each of which is defined as a program in execution. Because a process is a program, it can be represented as an algebra. Some of the major characteristics of a process include that it possesses its internal states and that it can communicate with other processes. Theoretical properties of communicating sequential processes (CSP) can be found in Hoare's book [18]. In this section, the discussion is limited to the process based design methodologies with emphasis on their design languages and graphical notations.

There are a number of design methodologies belonging to this approach. However, only two of them are discussed: JSD (Jackson System Development) [29–30] and MASCOT 3 (Modular Approach to Software Construction, Operation and Task) [31].

5.2.1. JSD

In JSD a system is viewed as a distributed network of processes, which interact with its environment. Jackson believes that a static world can be described in terms of a database. However, databases are not adequate enough for the description of dynamic worlds which are changing constantly, and sequential processes are recommended.

JSD consists of three major phases: the modeling phase, the network phase, and the implementation phase [25]. In the modeling phase the system under construction is modeled in terms of the major components which are considered as processes. Each process either performs some actions or suffers them.

JSD employs a natural language like English as its design language and a tree structure as its graphical notation. In order to illustrate this point, Cameron's library example [25] is used here: A library consists of the book, member, and reservation process. The book process is described using the English language and a tree:

ACTION	DEFINITION & ATTRIBUTES
ACQUIRE	The library acquires the book. id, date, title, author, ISBN, price.
CLASSIFY	The book is classified and catalogued. id, date.
LEND	Someone borrows a book. id, date, borrower.
RENEW	The borrower renews the book. id, date.
RETURN	The borrower returns the book to the library. id, date.
SELL	The book is sold. id, date, vendor, price.
OUTCIRC	The book is taken out of circulation as part of the inter-library swap scheme. id, date, destination
DELIVER	The book is delivered to the other library. id, date.

As one can easily see, JSD stands between the functional approach and the process-based in that its view of a system is close to that of SADT, its design language is informal, and the graphical notation is not so sophisticated as some of the design methodologies developed later. For the complete discussion of JSD, Jackson's book [26] is highly recommended.

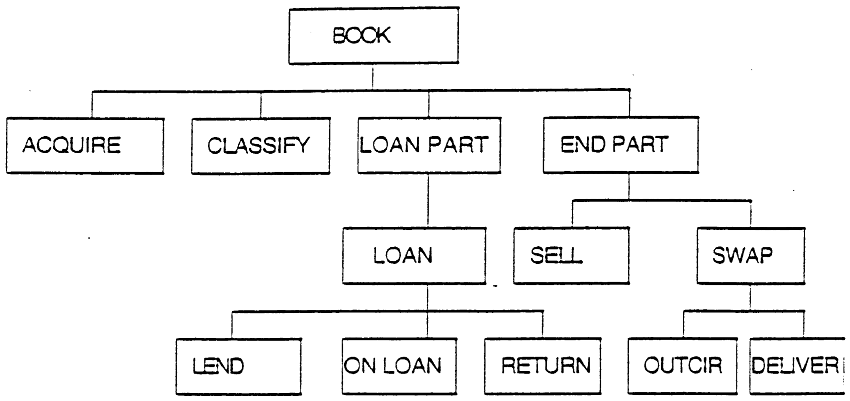


Figure 11. The description of the Book Process in English and as a tree.

5.2.2. MASCOT 3

Another very important design technique of this approach is MASCOT 3 [27]. It stands for Modular Approach to Software Construction and Operation and provides its design language and graphical notation. It has been used for the design of the software for large, distributed, embedded, real-time data processing system and has become a UK Ministry of Defense standard. In MASCOT 3, a system is regarded as a number of interconnected components which can be classified into two types of processes: activity and intercommunication data area (IDA). An activity process is represented by a round cornered rectangle, while an IDA is represented by a rectangle. To be specific, an example system is specified in terms of a diagram and its design language.

```
SYSTEM example_sys;
 USES subsys_1, subsys_2, subsys_3, sida_1, sida_2;
 IDA si1:sida1;
 IDA si2:sida2;
 SUBSYSTEM s2:subsys_2 (sp2=si1.sw,  ap2=si2.aw);
 SUBSYSTEM s3:subsys_3 (tp3=si1.tw,  rp3=si2.rw);
 SUBSYSTEM s1:subsys_1 (pp1=si1.pw, gp1=s3.gw3);
END.

SUBSYSTEM subsys_4;
 PROVIDES gw4: get;
 REQUIRES rp4:rec; otp4:out; tp4:trans;
 USES pool_1, chan_1, a_temp_1, a_temp_2;
  POOL p1:pool_1;
```

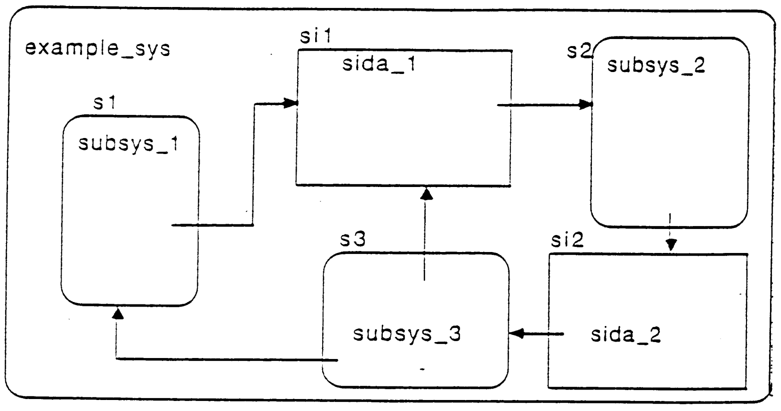


Figure 12. System example_sys.

```
  CHANNEL ch:chan_1;
  ACTIVITY a1:a_temp_1 (fp=ch.fw, tp=tp4, pp=t1.pw);
  ACTIVITY a2:a_temp_2 (sp=ch.sw, otp=otp4, rp-rp4);
  gw4=p1.gw;
END.

SUBSYSTEM subsys_3;
 PROVIDES gw3:get;
 REQUIRES rp3:rec; tp3:trans;
 USES subsys_4, serve_1;
 SERVERsv:serve_1;
 SUBSYSTEM s4:subsys_4(rp4=rp3, otp4=sv.otw,tp4=tp3);
 gw3=s4.gw4
END.
```

As illustrated above, the MASCOT design language is more precise and formal than those of SADT and JSD and closer to an algebra, and also its diagramming scheme is much more systematic than those of SADT and JSD.

5.3. Object-Oriented Approach

This is by far the most general and comprehensive approach incorporating techniques developed in both functional and process based approaches and is suitable for developing large, distributed, real-time, embedded and dynamic systems. There are a number of Object-Oriented system development methodologies, for example, OOD [7], MOON [32], HOOD

[33], ADM-3 [34], etc. All of them employ their own design languages and graphical notations, which are equivalent to algebras and diagrams in the categorical framework.

In this approach a system is viewed as a collection of objects. As pointed out earlier, two objects can be composed to form a new one. Inversely, an object can be decomposed. Furthermore, two or more autonomous objects can be linked together. Using the two operations on objects, a system can be built up from primitive objects and defined as a net of objects, i.e., system $= \langle$objects, nets$\rangle$. The graphical representation of the system is a diagram, where diagram $= \langle$nodes, arcs$\rangle$, where each node represents an object and an arc links two objects. Due to the fact that an object can be decomposed and the decomposed entities are objects again, the shapes of a node and an arc may change depending on what the node represents.

5.3.1. ADM 3 [30]

Of the object-oriented methodologies mentioned above, Firesmith's ADM 3 [30] provides the syntax and semantics of its design language and graphical notation, which may be transformed to algebras snd diagrams. To be specific, a diagram consists of nodes and arc, diagram $= \langle$nodes, arcs$\rangle$. ADM 3 provides designers with the freedom of defining and altering the shapes of nodes as well as the arc. rounded_rectangle, trapezoid). The ADM 3 design language is based on the assumption that a system consists of assemblies, sub-assemblies, and objects, and includes reserved words such as *assembly*, *subassembly*, *object*, *class*, *resource*, etc. Instead of reproducing the syntax of the ADM design language, an example is given in Fig. 15. The design language of ADM 3 is sufficiently algebraic and its diagramming scheme is well strucutured. In this sense ADM 3 may be viewed as a good realization of a category.

5.4. Net Based Approach

Ever since Petri introduced his net theory in the late 1960's, the Petri net has been employed as a tool for modeling computer controlled systems. A Petri net essentially consists of a set of *places* and a set of *transitions*, and is defined as follows [35]: A Petri net structure, PN, is a four tuple

$PN = \langle P, T, I, O\rangle$ where

$P = \{p_1, p_2, \ldots, p_n\}$ is a set of places, $n \geq 0$,

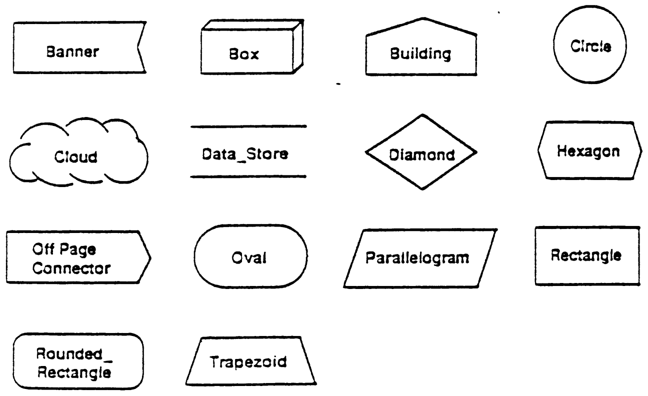
$T = \{t_1, \ldots, t_m\}$ is a set of transitions, $m \geq 0$,

$I\colon T \to P^\infty$ the input function,

$O\colon T \to P^\infty$ the output function.

When a Petri Net is applied to Object-Oriented Systems, the set of places can be replaced by the set of objects and the set of transitions can be replaced by the occurrences of events.

NODES = (banner, box, building, circle, cloud, data_store, diamond, hexagon, oval, off_papge_connector, parallelogram, rectangle, rounded_rectangle, trapezoid);



Arcs = (arrow, arrow_bent, arrow_both, arrow_clock, arrow_double, arrow_fork, arrow_reverse, arrow_triple, line_segment);

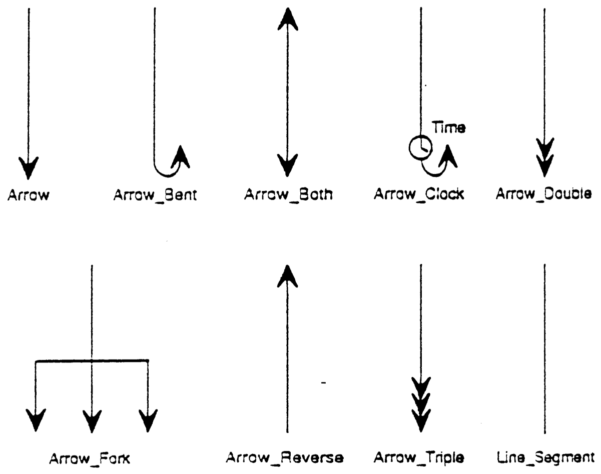


Figure 13. Different shapes of boxes and arrows in ADM 3.

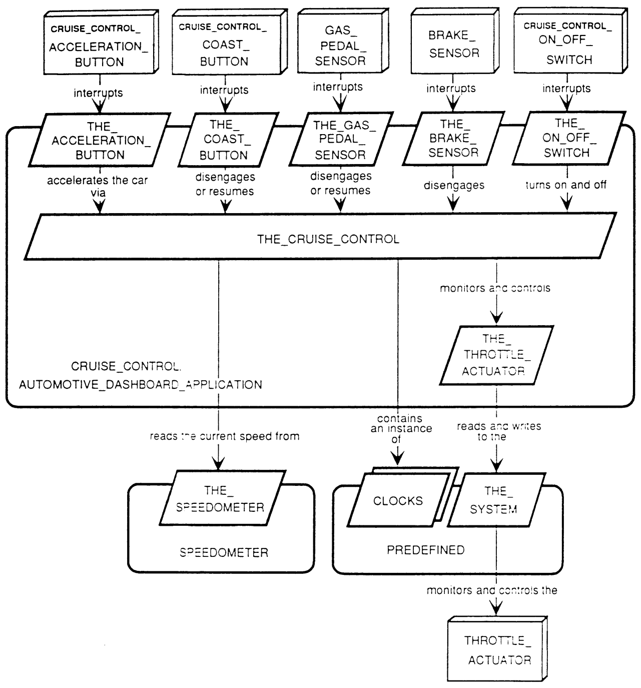


Figure 14. The diagram for CRUISE_CONTROL.

```
Object THE_CRUISE_CONTROL
 Parent Subassembly CRUISE_CONTROL;
 Specification
  Message ACCELERATE;
  Message DISENGAGE;
  Message RESUME;
  Message TURN_OFF;
  Exception FAILURE_OCCURRED_IN;
  Exception INCOMPATIBLE_PRIO_STATE_IN;
end;

Object THE_CRUISE_CONTROL
 Needs CLOCKS.PREDEFINED;
 Needs THE_SPEEDOMETER.SPEEDOMETER_IN;
body
 type Desired_speeds
 type States is (Accelerating, Disengaged, Maintaining, Off);
 variable The_Desired_Speed;
 variable The_State;
 modifier operation ACCERLATE;
 modifier operation DISENGAGE;
 preserver operation MAINTAIN;
 modifier operation ROUTE_MESSAGE_FOR
 modifier operation TURN_OFF;
 object THE_CLOCK;
start
 CONSTRUCT (THE_CLOCK).CLOCKS;
 ROUTE_MESSAGE_FOR;
 MAINTAIN;
end;
```

Figure 15. The specification of THE_CRUISE_CONTROL in ADM design language.

However, the original Petri net is good for modeling and simulation of concurrent systems, but not suitable for designing them because the Petri net lacks the notion of a hierarchy which is quintessential in the design of any system. For this reason, the concept of a hierarchy was introduced [36–37]. To the best of the author's knowledge, Reisig [38] was the first to use Hierarchical Petri Nets for the design of large systems.

In Reisig's method, a net consists of nodes and arrows, i.e. net $= \langle \text{nodes}, \text{arrows} \rangle$, where nodes represent objects and arrows represent relationships. There are two kinds of objects in this scheme: active and passive. A rectangle represents an active object, whereas a circle represents a passive object. An arrow, of course, represents a relationship between two objects.

Reisig defines two operations on objects: refinement and embedding. Refinement is equivalent to *composition*, while embedding is equivalent to *union* in our system.

5.5. An Observation

In this chapter various system design methodologies are classified into four distinct approaches, functional, process-based, object-oriented, and net-based, and are considered in terms of their views of a system, design languages and graphical notations, the latter two of which may be considered as algebras and diagrams in the categorical context.

One of the advantages of using the algebraic framework is that it allows one to extract essential features out of the "methodology jungle," put them in the proper perspective, and develop the much needed uniform design theory.

Due to the decomposability of an object, it may be decomposed into processes, each of which may be decomposed into operations or functions. For this reason, the object-oriented approach is based on the techniques associated with both process-based and functional approaches.

The following list summarizes the major views of a system discussed in the paper:

Functional Approach: System = ⟨Components, Relationships⟩

Process-based Approach: System = ⟨Processes, Interprocess Communication⟩

Object-Oriented Approach: System = ⟨Objects, Nets⟩

Net-Based Approach: System = ⟨Places, Transitions⟩

In addition, Fig. 16 lists the design languages and graphical notations of the major system development methodologies.

Combining theoretical and practical results on objects, the interactive system design tool, OBJ-NET, has been proposed and currently under development at the University of Michigan–Dearborn.

6. OBJ-NET [3, 4]

OBJ-NET is an interactive system design tool based on objects and nets. The theoretical basis of this tool comes from Category Theory in which objects and arrows (or equivalently algebras and morphisms) play the major role (see chapter 3). The theory provides the theoretical framework for object-oriented technology. In particular, as illustrated in the previous section, most software system design methodologies may be considered as special cases in the categorical framework.

In OBJ-NET, a system is viewed as a net of objects interacting with its environment. Each object can be decomposed into smaller ones and linked with other objects in terms of a net. Due to the two operations, *composition* and *union* on objects, a system may be specified as a hierarchy of objects as shown in Fig. 10, The major system components along with their

APPROACH		DESIGN LANGUAGE	GRAPHICAL NOTATION
FUNCTIONAL	SADT	NATURAL	DIAG =<BOXES, ARROWS>
PROCESS-BASED	JSD	NATURAL	TREE STRUCTURE
	MASCOT	PSEUDO CODE LIKE PASCAL	DIAG =<BOXES, ARROWS>
OBJECT-ORIENTED	MOON	PSEUDO CODE LIKE PASCAL	DIAG =<BOXES, ARROWS>
	HOOD	HOOD Design Language	DIAG =<BOXES, ARROWS>
	ADM	ADL	DIAG =<BOXES, ARROWS>
NET-BASED	PETRI NET	NATURAL	DIAG =<BOXES, ARROWS>

Figure 16. System design methodologies.

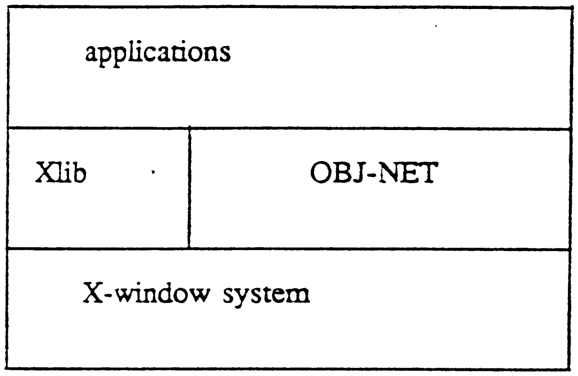


Figure 17. The OBJ-NET environment.

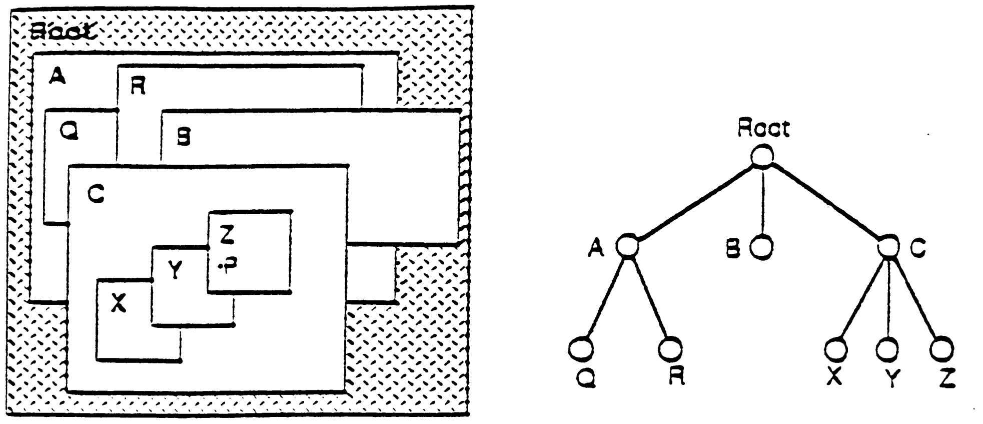


Figure 18. A three-level hierarchy of windows.

communication mechanisms are given below:

$$\begin{aligned}\text{system} &= \langle\text{subsystem, nets}\rangle\\ \text{subsystem} &= \langle\text{objects, messages}\rangle\\ \text{objects} &= \langle\text{processes, interprocesscommunicationmechanisms}\rangle\\ \text{process} &= \langle\text{modules, parameters}\rangle\end{aligned}$$

As one can easily see, the composition operation on objects induces the hierarchical relationship among objects, whereas the union (linking) induces the vertical relationship among them.

OBJ-NET has been implemented in the X-window environment. Reasons for choosing X-window are that it supports both hierarchical and vertical relationships and that it is portable. OBJ-NET sits on the top of the X-window system and interacts with application programs (Fig. 17). In OBJ-NET an object is represented as a window. The relationships among objects are equivalent to those among windows. In Fig. 18 windows A, B, and C are the children of the root, while windows Q and R are the children of window A and windows X, Y, and Z are the children of window C.

- **Architecture Option**

  When this button is clicked, the user is presented with the architecture menu which allows the user to specify the major components of a system uner development. The menu displays nine icons indicating system, subsystem, process, etc in the first column and seven functions across the top. The theoretical basis of this option is the algebraic framework presented in this paper.

- **Behavior Option**

  When this button is clicked, the user is presented with the behavior menu which has three options again: control, flow, and quit. The control option allows the user to specify the control rules among the objects and events, while the flow option lets the user to specify how control flows. The theory behind this option is based on Event-driven System Theory and Hierarchical Color Petri Net Theory.

- **Simulator**

  This option combines information provided by the Architecture and Behavior options and displays the flow of control by brinking lights as control flows from one object to another. After viewing a simulation, the user may return to architecture or behavior option and modify the design.

Further details on OBJ-NET may be found in Ref. 3 and 4.

7. Conclusion

Category Theory is introduced as the mathematical basis for Object-Oriented System Theory. As pointed out earlier, the theory can be applied to various areas of Computer Science. The advantage of Category Theory is that one does not have to master the entire field of Category Theory before applying it. In order to make this point and in order for mathematically unsophisticated readers to be able to employ the category-theoretical framework to their work, the minimum number of category-theoretical concepts are introduced in this paper.

The necessary categorical concepts have been re-interpreted in the Object-Oriented framework. Then an *open system* is defined as a net of objects interacting with its environment. Applying recursively the two operations *composition* and *union* on objects, the hierarchy of an open system has been established in Fig. 10. The hierarchy enables us to classify existing major system design methodologies into four distinct approaches: functional, process-based, object-oriented, and net-based. When they are examined in the algebraic framework, one can easily see that the notion of a system has been evolving from that of the functional approach to that of the net-based approach, that design languages have been developed from informal to formal (or algebraic) ones, and that diagrams have emerged as a systematic means of communication among end-users, designers, implementors, and maintainers of systems.

The immediate contribution of the algebraic or categorical framework might appear to be limited to the specification of the architecture of a system. On the contrary, by introducing Petri Nets as a special case of a category, one can easily incorporate the huge body of knowledge obtained from Distributed and Concurrent systems. However, this aspects of a system are beyond the scope of this paper and will be treated in a separate paper.

In this paper the application of Category Theory is limited to the design and specification of a system. However, the theory can be applied to various areas of Computer Science. The power of the categorical framework appears to lie in the fact that it provides us with mechanisms to integrate various activities in Computer Science.

References

1. D. H. H. Yoon, “The categorical framework of object-oriented concurrent systems.” *Computers and Mathematics with Applications: An International Journal* 25(2), pp. 33–38, 1993.
2. D. H. H. Yoon and L. S. King, “An object-oriented approach to computer integrated systems.” *Journal of Systems Integration* 6(3), 1996.
3. D. H. H. Yoon, Q. Zhu, and J. Cheng, “OBJ-NET: An object-oriented system design tool,” in *Proc. Int. Conf. on Modelling and Simulation*, Pittsbugh, PA, April 24–27, 1996, pp. 151–159.
4. D. H. H. Yoon, Q. Zhu, V. Mohanram, and J. Cheng, “OBJ-NET: An object-oriented system design tool.” To appear in *Journal of Systems Integration*.
5. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification I, II*, Springer-Verlag, New York, 1985.
6. R. M. Burstall and J. A. Goguen, “Algebras, theories and freeness: An introduction for computer scientists,” M. Broy, G. Schmidt, eds., *Theoretical Foundations of Programming Methodology*, D. Reidel Pub. Co., 1982.
7. G. Birkhoff and J. D. Lipson, “Heterogeneous algebras.” *Journal of Combinatorial Theory* 8, pp. 115–133, 1970.
8. A. Goldberg and A. Kay, *Smalltalk-72 Instruction Manual*, Palo Alto, CA, XEROX PARC.
9. G. Perry, *Moving from C to C++*, Sams, Indianapolis, 1992, Chap. 16.
10. A. S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, 1995, pp. 52–55.
11. G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, Menlo Park, CA, 1991.
12. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lrensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.
13. G. Agha and C. Hewitt, “Actors: A conceptual foundation for concurrent object-oriented programming,” in B. Shriver, P. Wegner, eds., *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA, 1987.
14. Y. Shoham, “Agent-oriented programming.” *Artificial Intelligence* 60, pp. 51–92, 1993.
15. P. Krief, *Prototyping with Objects*, Prentice Hall, New York, 1996.
16. O. H. Dahl and K. Nygaard, “SIMULA—An ALGOL-based simulation language.” *Comm. of ACM* 9(9), September, 1966, pp. 671–678.
17. *Webster’s Encyclopedic Unabridged Dictionary*, Portland House, New York, 1985.
18. R. M. Gallimore, D. Coleman, and V. Stavridou, “UMIST OBJ: A language for executable program specifications.” *The Computer Journal* 32(5), pp. 413–421, 1989.
19. B. C. Pierce, *Basic Category Theory for Computer Scientists*, MIT Press, Cambridge, MA, 1991.
20. M. Barr and C. Wells, *Category Theory for Computing Science*, Prentice Hall, New York, 1990.
21. M. A. Arbib and E. G. Manes, “Foundations of system theory: Decomposable systems.” *Automatica* 10, pp. 285–302, 1974.
22. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
23. E. W. Dijkstra, “The structure of the ‘THE’ multiprogramming system.” *Comm. of ACM* 11(5), May 1968, pp. 323–344.
24. C. Hewitt and P. deJong, “Open systems,” in M. L. Brodie, J. Mylogoulos, J. W. Schmidt, eds., *On Conceptual Modelling*, Springer-Verlag, New York, 1984.
25. D. C. Parnas, “A technique for software module specification with examples.” *Comm. of ACM* 15(5), pp. 330–336, 1972.
26. D. C. Parnas, “On the criteria to be used in decomposing systems into modules.” *Comm. of ACM* 15(12), 1053–1058, 1972.
27. S. Ginali and J. Goguen, “A categorical approach to general systems,” in G. J. Klir, ed., *Applied General System Theory*, Plenum Press, New York, 1978.
28. D. A. Marca, C. L. McGowan, and D. T. Ross, *SADT*, McGraw Hill, St. Louis, 1988.
29. J. Cameron, “An overview of JSD.” *IEEE Trans on Software Engineering* SE-12(2), February 1986.
30. M. Jackson, *System Development*, Prentice-Hall, Englewood Cliffs, NJ, 1983.
31. G. Bate, “MASCOT 3: An informal introductory tutorial,” *Software Eng. J.* 1(2), 1986.
32. M. Hall, P. O’Donoghue, and B. Hagan, “MOON—Modular object-oriented notation.” *Software Engineering Journal* SE-6(1), January 1991.
33. B. Delatte, M. Heitz, and J. F. Muller, *HOOD*, Reference Manual 3.1, Prentice Hall, London, 1993.
34. D. G. Firesmith, *Object-Oriented Requirements Analysis and Logical Design*, John Wiley & Sons, New

York, 1992.

35. J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, NJ, 1981.
36. P. Huber, K. Jensen, and R. M. Shapiro, “Hierarchies in Coloured Petri Nets,” in K. Jensen, G. Rozenberg, eds., *High-Level Petri Nets: Theory and Application*, Springer-Verlag, New York, 1991.
37. V. O. Pinci and R. M. Shapiro, “An integrated software development methodology based on hierarchical colored Petri nets,” in G. Rozenberg, ed., *Advances in Petri Net 1991*, LNCS vol. 524, Springer-Verlag, NY, 1991.
38. W. Reisig, *A Primer in Petri Net Design*, Springer-Verlag, New York, 1992.