Journal of Systems Integration, 6, 159–179 (1996)

An Object-Oriented Approach to Computer Integrated Systems

D. H. H. YOON
Department of Computer & Information Science, University of Michigan—Dearborn, Dearborn, MI 48128

L. S. KING
Industrial & Manufacturing Systems Dept., GMI Engineering & Management Institute, Flint, MI 48504-4898

Received July 7, 1993; Revised August 8, 1995

Abstract. In recent years computers have been incorporated into large scale systems such as nuclear plant, flight control, and manufacturing systems. Such Computer Integrated Systems (CIS) normally consist of heterogeneous subsystems. The integration of heterogeneous subsystems requires that the subsystems be portable, inter-operable, and integrable at both software and hardware levels so that the integrated system should function properly. Objects and nets are proposed as the atomic elements of CIS's. An object is defined as a computational model of an arbitrary entity. Then three representation schemes of an object are introduced: algebraic, modular, and graphical. Two operations on objects, *Composition* and *Union*, are introduced as means of combining two objects into a larger one. As an application of this approach, a Computer Integrated Manufacturing (CIM) system is represented as a network of objects.

Keywords: Computer Integrated Systems, objects, nets, algebras, layers, computer controllers, CIM

1. Introduction

Computers have been employed in all walks of life ranging from small offices to large factories manufacturing automobiles, aircraft, and ships. As a typical Computer Integrated System, we consider a computer information system of a corporation which consists of four departments: Administration, Marketing and Sales, Engineering, and Manufacturing. The first two departments might use a Management Information System (MIS) and an Office Information System (OIS), respectively, while the Engineering department employs a Computer Aided Design (CAD) system and the manufacturing department has a Computer Integrated Manufacturing (CIM) system. For the efficient operation of the company, the subsystems need to be interconnected, and information should be able to flow without barrier from one department to another. From the systems standpoint, the company's information system is a network of autonomous subsystems (Fig. 1).

Of the four departments, the manufacturing department involves most intricate aspects of CIS's. In particular, a manufacturing subsystem consists of a number of workstations: a transport mechanism, loading/unloading machines, and a controller. The term 'workstation' in Manufacturing should be distinguished from a computer workstation: It refers to a group of machines such as robots, sensors, and a computer which perform unique services on raw material or parts on the transport system (Fig. 2).

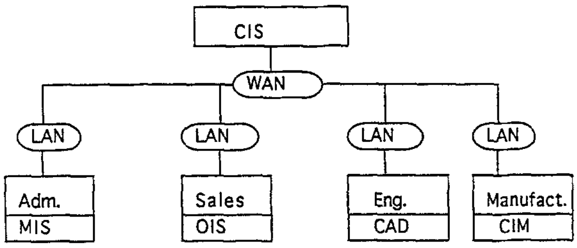

Figure 1. A corporate information system.

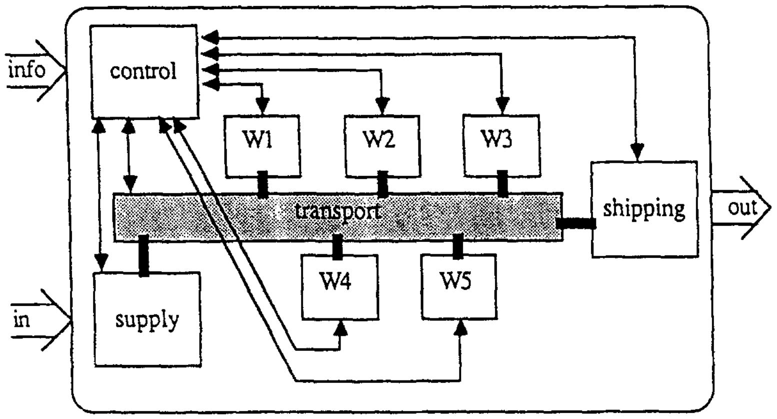

Figure 2. A typical manufacturing cell.

The information system presented above demonstrates major characteristics of CIS's:

1. A CIS consists of heterogeneous subsystems which have been designed and manufactured according to different design principles by different vendors.

2. One or more components of a CIS can be added or deleted depending on the needs without damaging the integrity of the entire system.

3. Each subsystem is autonomous and capable of communicating with other subsystems.

4. A CIS could merge into a larger system.

When computers are employed as controllers, all the components of the system should be represented inside the computers. For this purpose, we propose objects as the computational model. In section 2, major characteristics of a CIS are discussed in depth. In section 3, we formalize the notion of an object and discuss its representation techniques. In section 4, a CIM system is presented applying the theory developed in section 3.

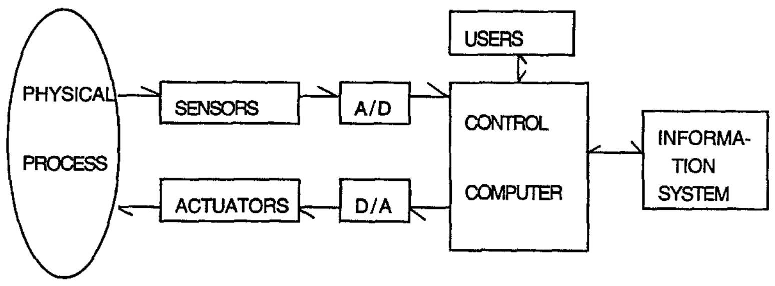

Figure 3. A computer controlled workstation.

2. Computer Integrated Systems (CIS)

Computers have been employed as controllers of various systems: chemical and petrochemical plants, oil refineries, iron and steel plants, power plants, etc. With rise of recent industrial automation and flexible manufacturing, the importance of computers in large systems has been escalated. In this section, the primary functions of a control computer are identified. And then objects are proposed as the primary tool for designing necessary control software.

In manufacturing environments, a control computer often collects data through sensors, analyzes it, and takes corrective actions on actuators. In addition it displays the system status on the screen for human users, while maintaining the data and knowledge bases.

Fig. 3. pictorially summarizes the primary functions of a control computer in manufacturing environments. Interfacing with sensors and actuators is one of the oldest problems. There are many kinds of sensors, for instance, position, vision, touch, force, temperature sensors and various actuators such as electrical drive and binary actuators [1,2]. Interfacing with electronic devices is pretty well understood at hardware level. However, the problem is yet to be resolved at software level and requires further research.

Another type of interface deals with human users. The control computer not only displays the status of the system on the screen, but also communicates with human users. In particular, a human operator should be able to start and stop the process and alter the flow of control by issuing commands interactively. These operations should be done in real-time. Most work on graphical user interface design deals with office environments [3] and designing graphical user interfaces for manufacturing environments is a wide open area.

In addition to interfaces, manufacturing database is quite different from its classical counterpart in that there could be many versions of a data model in the system. For instance, one version might be in a manufacturing cell, while another version representing the same entity could be in a pipeline and they may not agree with each other. One of the most important problems in manufacturing database is to maintain the uniformity of data throughout the system [4].

The guiding principles of the CIS software are derivable from those of operating systems because computers control the entire systems. Dijkstra [5] defines a system as a society

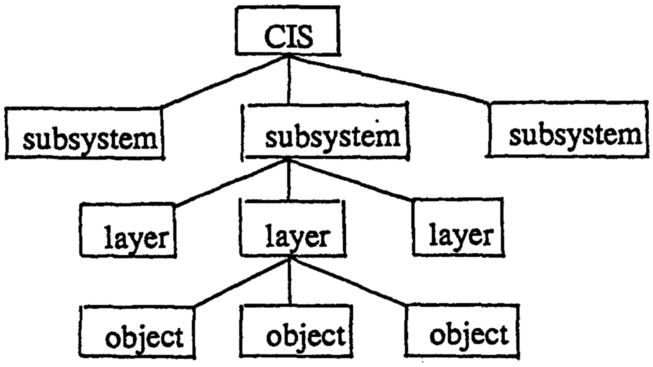

Figure 4. The hierarchical structure of a CIS.

of communicating sequential processes (CSP), while Hewitt [6] views it as a "society of communicating experts." In a way Dijkstra's CSP and Hewitt's expert are very similar and their concepts are embedded into Ginali & Goguen's object [7]. In this paper a CIS will be viewed as a network of objects which are computing agents capable of communicating with other agents [8].

Objects are the fundamental elements of software systems and grouped into a layer. Layering is so crucial that it distinguishes CIS's from the classical systems. In fact, it is the very essence of CIS's that makes them organic and flexible. It is not a coincidence that most of the historically important operating systems have been designed in layers: Dijkstra's 'THE' [5] uses five layers, Liskov's Venus [9] is in six layers, Multics [10] employs multiple ring structure, and UNIX [11] has been designed in three basic layers. Layered architecture culminates in the design of Open Systems Interconnect (OSI) reference model [12] which appears to be the standard of CIS's. A CIS can be succinctly described in a hierarchical structure as in Fig. 4.

Such object based systems are inherently distributed in that each major component has its own hardware and software, parallel in that two or more components perform operations in parallel, and real-time in that operations are time-critical.

3. The World of Objects

As the tree in Fig. 4 indicates, objects are the atomic elements of CIS's. The application of objects is not limited to system design. They have been employed in just about all branches of computing: languages, database, software engineering, etc. However, there is no standard definition of an object. In this section, we trace how the notion of an object has emerged in the history of computing and define it precisely as a computational model for any entity that needs to be represented in a computer. Then two operations on objects, *composition* and *union*, are introduced as means of combining two or more objects into one larger one.

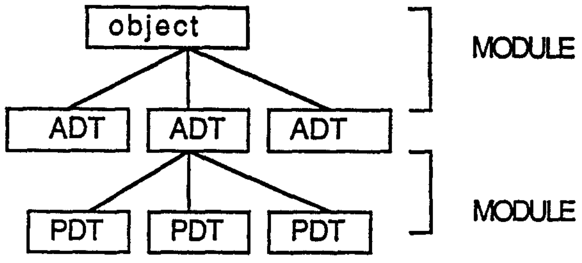

Figure 5. The hierarchy of computational models.

3. i. *The Emergence of Objects*

Objects were initially introduced and applied in the design of Simula [13]. However, the concept of an object had been around throughout the history of computing. Booch [14] classifies programming languages into five generations: The first generation programming languages (1954–1958) including Fortran and Algol 58 dealt primarily with mathematical objects, most of which are implemented as primitive data types (PDTs) in modern programming languages. The second generation languages (1959–1961) including Fortran II and Algol 60 introduced modules and modularity to overcome some shortcomings of the first generation languages. The third generation languages (1962–1970) such as Pascal, C, and Simula, introduced abstract data types (ADT), which have become the precursors of objects. It turns out that PDT's and ADT's are just special cases of objects.

To be specific, consider the following representations of mathematical objects: boolean = $\langle B, \{\text{and}, \text{or}, \text{not}\}\rangle$, character = $\langle C, \{\text{ord}, \text{chr}, \text{pred}, \text{succ}\}\rangle$, integer = $\langle Z, \{+, -, *, \text{div}\}\rangle$, where B, C, and I represent the set of booleans, characters, and integers, respectively. Notice that each mathematical object is represented as a set with operations defined on it. The same format is used to represent ADT's: stack = $\langle$array, {push, pop}$\rangle$ and queue = $\langle$array, {insert, delete}$\rangle$.

One can easily see the structure underlying the two groups: An entity is represented as an encapsulation of data and operations defined on it. Generalizing the above, an object can be defined as a computational model of an arbitrary entity and will be represented as a set of data and operations defined on it. Being a computational model, an object will be implemented in a particular programming language. To be precise, a programming language provides a set of primitive data types (PDT) such as integer, real, character, boolean, etc along with operations on them. A user can define ADT's in terms of PDT's and modules so that he may be able to represent an entity that requires a more sophisticated structure than that of PDT's. As the hierarchy indicates (Fig. 5), an object has emerged as the most general computational model and is based on ADT's and PDT's.

```
OBJ stacktype;
        SORTS item; stack;
        OPNS
                push: stack * item --> stack;
                pop : stack --> item;
        EQNS
                for all i ∈ item, s ∈ stack,
                push(s,i) --> s;
                pop(s) --> i;
ENDOBJ.
```

Figure 6. The algebraic specification of the stacktype.

3. ii. *Three Representation Schemes of Objects*

In this section, three object representation techniques are presented: Algebraic, modular, and graphical. The three are essentially equivalent, but their levels of abstraction are different.

The algebraic representation is highly abstract and results from an effort to transcend implementation detail. This technique is based on the notion of an algebra introduced by Birkhoff & Lipson [15]. A little later Goguen et al (known as the ADJ group) [16] have applied initial algebras to specify ADTs, and initiated the area called Algebraic Specification.

The modular representation is equivalent to the algebraic one, depends on particular programming languages and includes the *class* mechanism in C^{++}, the package in ADA, and the actor in the Actor languages as special cases. Modules were introduced during the second generation programming language design period and their concept was consolidated by Parnas [17,18], and has recently been further formalized by Ehrig and Mahr [19].

The graphical representation of an object is introduced as design aid. There are numerous schemes. However, most of them can be viewed as variations of a graph consisting of vertices and edges. Later in this section some popular techniques are briefly discussed.

3. ii. a. *The Algebraic Representation*

In order to introduce the notion of an algebra painlessly, the object stack is specified in an algebraic form first. Then the precise definition of an algebra follows.

The stack example in Fig. 6 is specified in OBJ2 [20] in order to introduce the notion of an algebra and also terminologies needed in the precise definition of an algebra. OBJ stacktype is an algebra which consists of *sorts* and *operations*. A sort is equivalent to a data type, whereas the operation section, denoted by OPNS, is equivalent to the declaration section of a programming module in which all the variables and procedure declarations are included and also equivalent to the signature of an algebra. The EQNS section specifies the semantics of all entries in the OPNS section. Using the intuitive meanings of a *sort* and *signature*, an algebra is formally defined as follows:

Definition. 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 $\{f \in \Sigma\}$ such that if arity $(f) = n$, then f_S: $S_1 \times S_2 \times \cdots \times S_n \to S_k$ where $1 \leq k \leq n$.

One can easily see that the stacktype specified above is an algebra with S = SORTS, Σ = OPNS, and Σ_S = EQNS.

The power of the algebraic specification lies in the fact that all major concepts in Computer Science such as PDTs, ADTs, processes, and objects can be represented as algebras. Furthermore, a new algebra can be obtained by taking a sum or product of existing algebras. This is essentially equivalent to obtaining a new object by taking either a composition or union of existing objects. Algebras with homomorphisms form a category which provides the theoretical basis of Distributed Computing in which objects communicate with one another through communication channels [8].

3. ii. b. *The Modular Representation*

There are a number of programming languages which provide mechanisms for representing objects. An object can be represented as a 'class and object' in SIMULA, a 'module' in Modular-2, a 'package' in ADA, or a 'class' in C^{++}. All these entities are equivalent to an algebra introduced in the previous section. In this section, the object stack is represented as an instance of the class stack in C^{++}. And then we will establish the equivalence between an algebra and a class in C^{++}.

A class consists of two sections: private and public. The private section includes variables used on the class, whereas the public section includes the name of functions, which are known as member functions and allowed to access the data declared in the private section. Other objects can access the private section of an object only through its member functions. The explicit semantic definitions of the member functions follow the public section. For the syntax of C^{++}, the reader is referred to Stroustrup's book [21].

There is a one-to-one correspondence between a class and an algebra: The private section is equivalent to the set of sorts S in the definition of an algebra, the public section is interpreted as the signature Σ, and the rest can be viewed as the semantics of the signature Σ.

```
const max_len = 1000;

class stack {
      private:
            char s[max_len];
            int top;
      public:
            stack (int size);            // the constructor
            ˜ stack () ;                 //the destructor.
            void push(char c);
            char pop();
          }

stack::stack(int size)
      {
      s=new char[size];
      top=0;
      }

void stack::push(char c)
      {
      top++;
      S[top]=c;
      }

char stack::pop()
      {
      return s[top--];
      }
```

Figure 7. The object stack in C^{++}.

3. ii. c. *The Graphical Representation*

The two object representation techniques, algebraic and modular, are useful when fine detail is desired. However, at the design stage too much detail is rather cumbersome than helpful. In a situation like this, graphical representation techniques have been proven effective.

There are a number of graphical representation techniques proposed and in use: SofTech's SADT (Structured Analysis and Design Techniques) [22], the MASCOT (A Modular Approach to Software Construction, Operations and Test) series [23], HOOD (Hierarchical Object Oriented Design) [24], and ADL (ASTS Diagramming Languages) [25]. These techniques essentially represent objects and the static relations among them using boxes and arrows. However, Reisig [26] employs Petri nets to describe the dynamic behavior of a system, a network of objects.

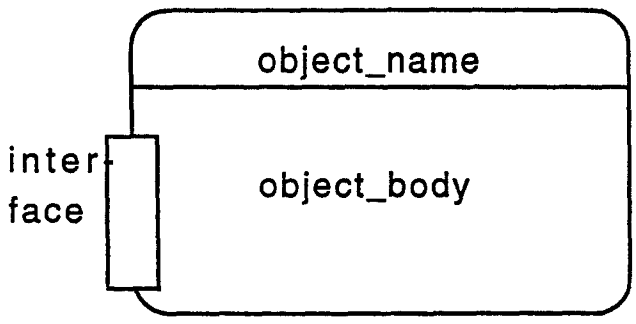

Figure 8. A HOOD box representing an object.

An object can be extremely complex and possesses many attributes. In this section, some major properties of an object are briefly summarized using the HOOD representation: An object has a name, a body, and an interface (Fig. 8). The body of an object encapsulates local data and operations defined on the data and hides its internal operations from other objects, whereas the interface provides a mechanism for exchanging information with other objects.

Most graphical representation techniques of objects can be explained in terms of graphs consisting of vertices and edges, i.e., $G = \langle V, E \rangle$. Indeed, graphs form a basis for visualizing not only objects, but also information [27].

3. ii. d. *Operations on Objects*

Earlier in this paper, it was pointed out that a new object can be obtained by combining existing objects. This can be achieved through the composition and the union operations on objects.

3. ii. d. A. *Composition*

The composition of two objects stems from the exporting and importing capacities of objects and is equivalent to an external procedure call in procedural languages. To be more specific, let us consider the following examples:

In the above example, two objects $O1$ and $O2$ export themselves so that other objects can utilize their services. The object $O3$ imports two objects $O1$ and $O2$ using the phrase 'USES' for their services. This will be interpreted as the composition of two objects, $O1$ and $O2$, and is denoted by $O3 \supset O1 * O2$. Its graphical representation is given in Fig. 10.

```
OBJECT O1 (par1)                  OBJECT O2 (par2)
      SORTS ....;                       SORTS ...;
      OPNS -----                        OPNS ----
            -----                             ----
      EQNS ----                    EQNS ----
            ----                              ----
ENDO1;                            ENDO2;


OBJECT O3 (par3)
     USES O1, O2;
       SORTS ....;
       OPNS -----
             -----
       EQNS ----
             ----
ENDO3;
```

Figure 9. A composition of two objects in O3.

3. ii. d. B. *Union*

The union operation provides a mechanism to connect two or more independent objects in terms of nets. A typical example of the union of two objects is the client-server model in which the client is the object requesting service from the server object. In this model, the client and the server are assumed heterogeneous in the sense that their architectures are distinct and communicate with each other by sending messages only.

Since the client and the server are heterogeneous, they speak different languages. Hence there is a need for a universal language that both client and server understand. When the client attempts to send a message to the server, the client's translator translates the message into the universal form. The message is transmitted through the network link and reaches the server's translator which translates it into the server's vernacular.

In a large system, it is very common that there are various clients and servers. This requires that translators be robust so that they can handle various dialects. Responding to this need, ISO (International Standards Organization) has proposed OSI (Open Systems Interconnection) reference model [28], which consists of seven layers: application, presentation, session, transport, network, data-link, and physical. The translator in our model incorporates the five intermediate layers except the first and the last layers.

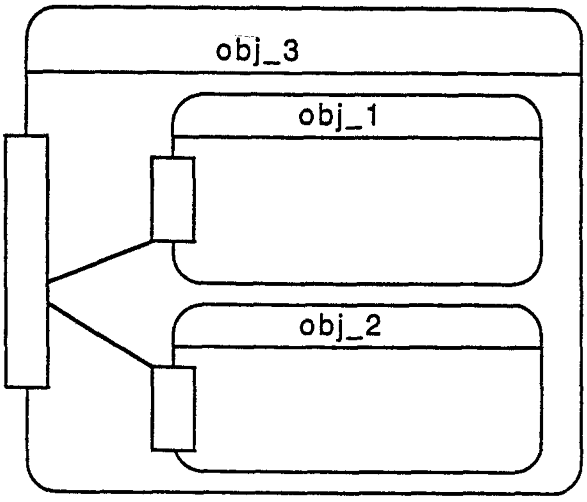

Figure 10. The graphical representation of the composition.

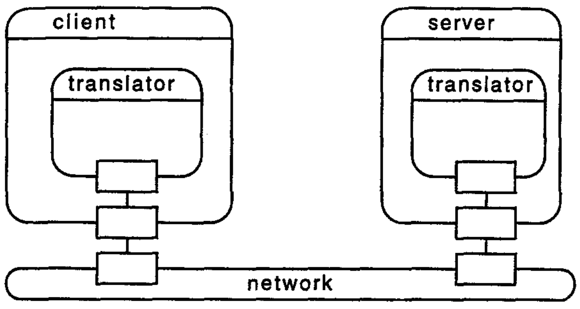

Figure 11. The client-server model.

3. ii. e. *Open Systems*

Putting together what has been discussed in previous sections, an open system is defined as the union of objects and viewed as a composite object (Fig. 12).

Each object in the system is considered as an autonomous computing agent capable of communicating with other agents in the sense that it has its own hardware, software, and an interface mechanism. Since each object possesses its own hardware and software, communication is achieved by sending and receiving messages only. Hence, the function of the interface is crucial. The interface was introduced as a translator in the previous section (Fig. 11). It will be further elaborated based on the ISO's Reference Model of Open System Interconnection (OSI).

The OSI Reference Model consists of seven layers: application, presentation, session,

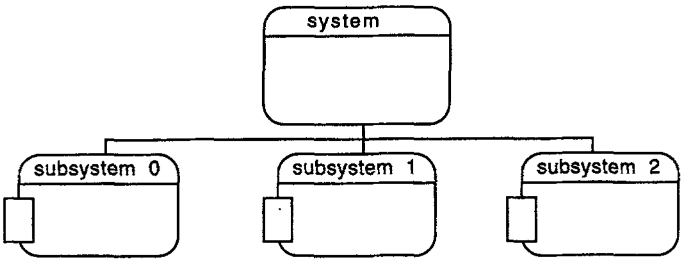

Figure 12. A system as a network of objects.

transport, network, data-link, and physical. The primary functions of each layer is as follows:

1. The physical layer is concerned with the transmission of a raw bit stream between two objects.
2. The data-link layer introduces reliability by providing functions for recovering from transmission errors.
3. The network layer breaks a message into packets and controls both routine through the network and congestion in order to provide for high performance.
4. The transport layer provides reliable host-to-host communication and network independence by hiding the details of the communication network.
5. The session layer manages process-to-process communication.
6. The presentation layer provides the facilities for commonly performed data transmission.
7. The application layer consists of the collection of user programs.

When a client object sends a message to a server, the message passes down the layers on the client's side, goes through the physical layer, and finally passes vertically up the layers on the server's side. Each layer at the sending side performs some functions on the message, attaches a header, and sends the packet to the next layer. On the server's side, each layer removes the header associated with its level and performs the necessary work based on this information [29].

In an actual implementation, many of the functions found in the bottom three layers of this model are likely to be placed in hardware, while the remaining layers are typically software functions. Most applications may not need all seven layers and consequently may reduce the total number of layers by combining the proposed layers.

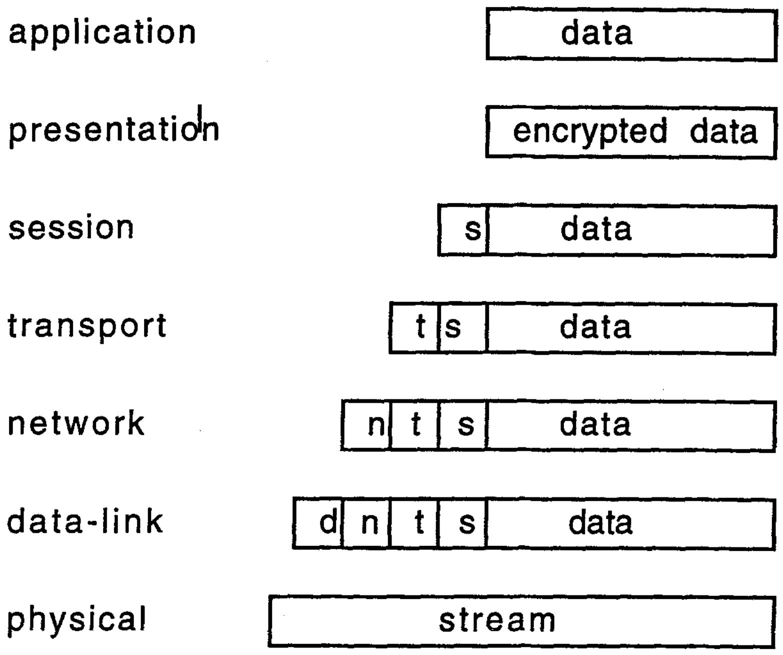

Figure 13. Data formats.

4. A Computer Integrated Manufacturing System

As an illustration of the theory developed thus far, a CIM system is presented as a net of objects. Fig. 14 illustrates the manufacturing system at GMI [30] which consists of a few workstations, a transport system at the center and the palletizing unit. The robot at the palletizing unit places materials or parts on the conveyor system. The parts are processed first by the lathe in workstation 1 and then in the milling stations (workstations 2 and 3). When the processing is completed, the product is removed from the conveyor belt by the robot in workstation 4.

As Fig. 15 illustrates, the control structure of the manufacturing system is hierarchical. Two workstations form a workcell and are controlled by a cell controller. Several cells constitute a shop floor and are controlled by a floor controller. Computers are employed as controllers at all levels. The workcell controller is responsible for communication with other cell controllers and for controlling devices in the cell. The shop floor controller, on the other hand, communicates with other offices, lab controllers, and all the cell controllers under its supervision.

The CIM system presented above consists of three broad categories of objects:

$$\text{CIM} = \{\text{subsystems}\} + \{\text{nets}\} + \{\text{messages}\}$$

The subsystems include the conveyor system, robot-arms, milling machines, lathes, and

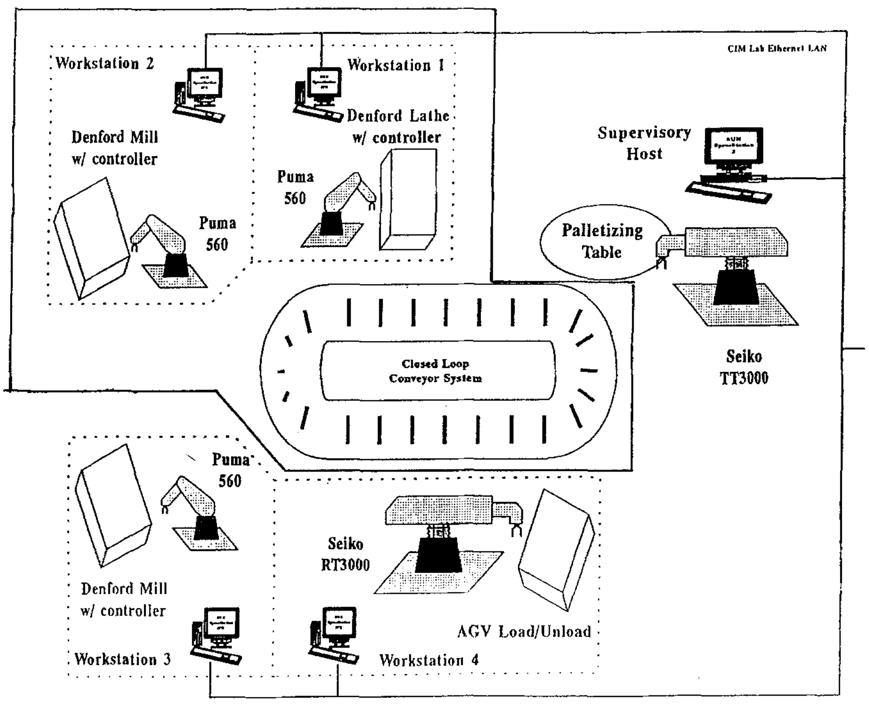

Figure 14. The layout of a manufacturing cell.

parts, whereas the nets could be either local or wide area nets. In the CIM lab at GMI, ethernet is employed. The messages in the manufacturing environments will follow the MMS (Manufacturing Message Specification) format specified by ISO [31].

The components are normally represented as virtual manufacturing devices(VMD) in manufacturing literature, which are equivalent to objects in our theory. Each object is an instance of the class. For example, Puma 560 is a particular instance of Object Robot-arm whose primary task is to pick an object and place it at a specified location. The advantage of representing components by classes is that particular devices can be replaced by different brands without affecting software code. Similarly, milling machines and lathes are instances of Object machines whose primary functions include cutting and drilling.

Each workstation is capable of transporting, machining, assembling and material handling. In particular the palletizing unit puts raw materials on pallets and places them on the transporting system. The lathe in Workstation 1, upon receiving a piece of raw material, mills its faces and sides. The milling machine in Workstation 2, upon receiving surfaced pieces from workstation 1, mills base slots and pockets, drill holes, and engraves letters on surfaces. Workstation 4 is responsible for assembling knobs and covers, putting covers on bases, and finally packaging the finished products in cartons.

Fig. 16 illustrates desk-top sets manufactured by students in the second author's CIM

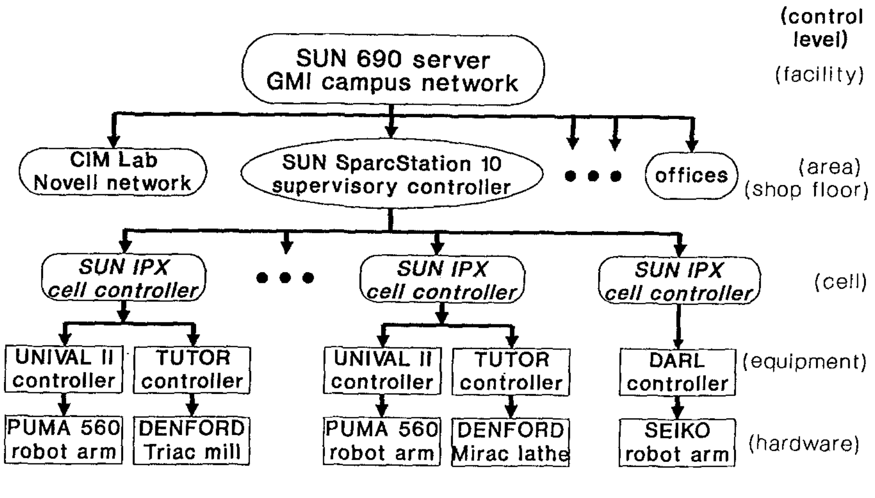

Figure 15. The hierarchical control of the cell.

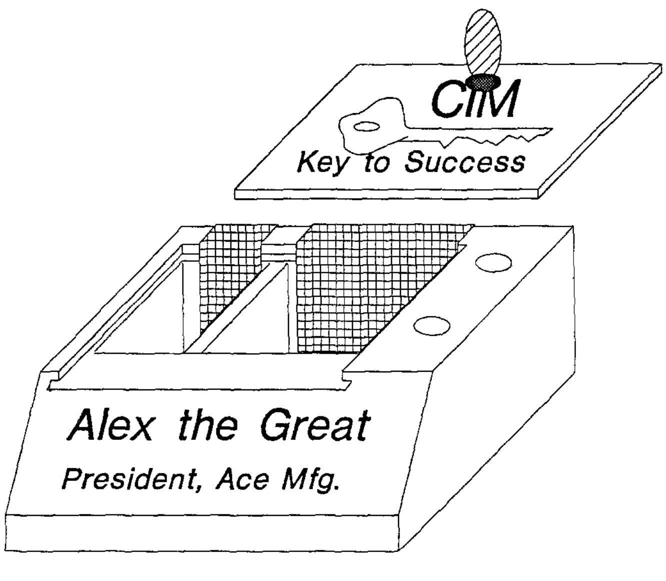

Figure 16. A desk-top set manufactured in the GMI CIM lab.

class at GMI using the above system. The large pocket on the right is for business cards, the smaller one on the left is for post-it papers, the holes on the side are meant to be for paper clips and rubber-bands. Raw materials required for this project include machinable wax blocks, plastic plates, plastic dowels, and wood for the pallets on AGV trays.

Each component introduced above is an object in our theory, communicates with other

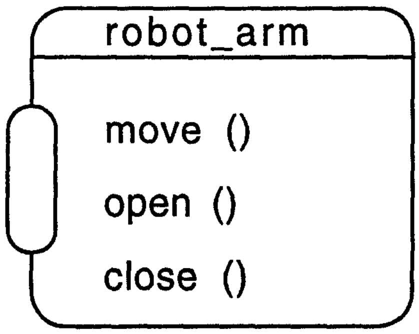

Figure 17. A robot-arm.

objects, and performs a specific task. In order to illustrate the notions of the algebraic and the graphical representation of an object, OBJ robot_arm is specified algebraically and graphically below.

4. i. *Object Robot-Arm*

A robot arm typically picks an object and places it at a specified location. This is achieved by moving the gripper to a specified position, opening the gripper, closing it, moving the object to a specified location, and opening the gripper.

```
OBJ robot-arm
    OPNS & EQNS
        move (p) which moves the gripper to the position p;
        open() which opens the gripper;
        close() which closes the gripper;
ENDOBJ.
```

Other manufacturing devices can be specified in a similar manner. For instance, milling machines and lathes belong to the same class whose primary function is cutting. The specific cutting operations such as face milling, side milling and face-side milling, are pre-programmed. From the control stand-point, the controller initiates and terminates particular cutting operations. Hence, OBJ cutting_machines can be specified in terms of the start() and stop() operations, which is suppressed here.

In the remainder of this section, the client-server model and the message object are specified.

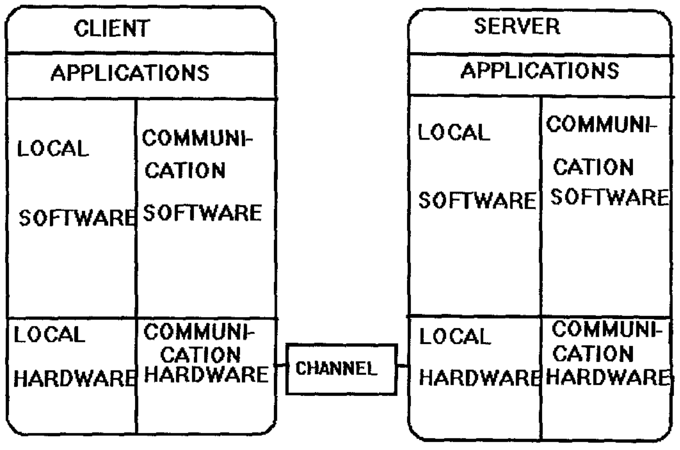

Figure 18. An object with a communication subsystem.

4. ii. *The Client-Server Model*

In section 3. ii. *union* was introduced as an abstract operation linking two objects. In this section, the client-server model is introduced as the vehicle for realizing the abstract operation *union.* The model has been discussed a great deal in literature without reference to its eventual merging into larger models. This creates the impression that the model exists in isolation. However, the primary function of the client-server model is to link two autonomous objects which provide distinct services. For example, a cell controller and a milling machine need to be connected so that they may be able to communicate with each other in order to achieve specific tasks. For this reason, the linking aspect of the client-server model is emphasized.

The client-server model requires its own hardware and software. To be specific, a typical object has its own hardware and software to provide service to other objects and also requires, for communication, a communication system consisting of its own interfacing hardware and communication software (Fig. 18). In order to stress the significance of both the hardware and software aspects of the model, the communication system of an object is further elaborated in Fig. 19.

The specification of the client-server object thus acts as a contract between the client and the server, which declares the responsibilities of each. It is the client's responsibility (a) to send the message to the right server, (b) to send only messages that are declared in the server's specification, and (c) to supply compatible parameters. It is the server's responsibility that (a) to check the compatibility of parameters, (b) to ensure that the correct operation receives the message, and (c) to correctly perform the operation.

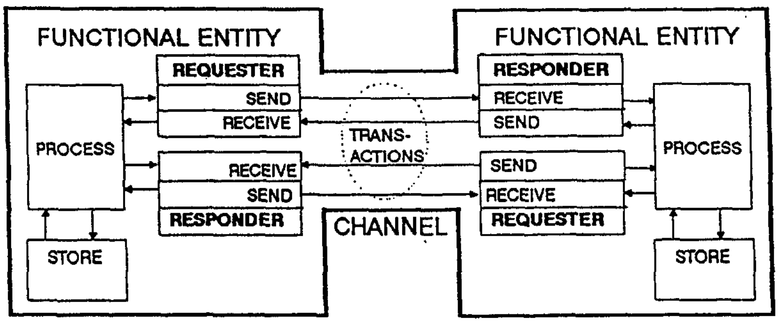

Figure 19. Hardware and software of client-server model.

Based on the diagram in Fig. 19, the communication subsystem can be specified as below. An object can be a client or a server. The only difference between the client and the server is the order in which sub-objects *requester* and *responder* are executed.

```
OBJ requester                   OBJ responder
   OPNS & EQNS                        OPNS & EQNS
      send();                              receive();
      receive();                           send();
OBJ requester                   ENDOBJ.
```

```
OBJ client                      OBJ server
   USES requester, responder;      USES requester, responder;
   OPNS & EQNS                        OPNS & EQNS
      requester;                      responder;
      responder;                      requester;
ENDOBJ.                         ENDOBJ.
```

4. iii. *Messages*

In 3. ii. e, a system was introduced as a collection of heterogeneous component objects which communicate with one another. And also it was pointed out that the communication subsystem consists of seven layers and that each layer performs some operations on messages. In this context a message is defined as a list of items, each of which can be added to the list or deleted from the list.

To be specific, let's assume the following:

create (m) = creates the message 'm' as an empty message,

add (i, m) = appends the element 'i' to the beginning of the message 'm', delete (m) = deletes the far-left element of the message 'm'.

Using the above, the object message is specified as follows:

```
OBJ message
      OPNS & EQNS
            for all i ∈ items; m ∈ message
            create() —> message;
            add(i,m) -> message;
            delete (m) —> message;
ENDOBJ.
```

Composing the above objects, client, server, and message, OBJECT comm is specified as follows:

```
OBJ comm
      USES client, server, message;
      OPNS & EQNS
            case i of
                  0: client;
                  1: server;
            endcase;
ENDOBJ.
```

As one can easily see, OBJ comm is obtained as the composition of three objects, client, server, and message and will function as the atomic element of the network. The specifications of the objects in this section have been intentionally kept simple for clarity. The above specifications need to be further refined in order to be used in the design of actual systems.

5. Conclusion

Objects have been introduced as the fundamental elements of the new mode of computing, generalizing the notions of PDT's, ADT's, processes, and prototypes. They function as the atomic elements of Open Systems. A CIM system is illustrated as an open system.

In this paper three object representation techniques have been introduced: algebraic, graphical, and modular. The algebraic representation is based on the notion of a heterogeneous algebra and forms the basis of the much needed theoretical foundation of object based computing. The graphical representation has been developed as a design tool, yet has to be unified and formalized. Graph Theory is proposed as the theoretical basis for the graphical representation of objects and includes Petri nets as a special case. Finally the modular representation technique involves all object-oriented programming languages: Ada, C^{++}, Modula, Smalltalk, etc. It was the intent of this paper to transcend the limitations of the programming languages so that the essence of object based computing is emphasized.

As indicated in the introduction, computers are employed just about in all aspects of our lives and objects will play the major role. It is imperative to develop the firm theoretical foundation of object based computing, which will provide the basis for both applied and theoretical work. Category Theory is proposed as the theoretical basis of object based computing, in which objects are represented as algebras and the relationships among the objects are represented as morphisms. In Category Theory, graphs are commonly employed as the means of visualizing categories consisting of objects and morphisms. This aspect will be further discussed in a separate paper.

References

1. W. J. Tomkins, J. G. Webster eds, *Interfacing Sensors to the IBM PC.* Prentice Hall: Englewood Cliffs, NJ, 1988.
2. Y. Koren, *Computer Control of Manufacturing Systems.* McGraw-Hill: New York, 1983.
3. J. Foley, V. Wallace, P. Chen, "The human factors of computer graphics interaction techniques." *IEEE Computer Graphics & Appl.* pp. 13–48, Nov. 1984.
4. J. Schwarz, B. Westfechtel, "Integrated data management in a heterogeneous CIM environment." in *Computers in Design, Manufacturing, and Production, 1993 CompEuro Proceedings*, A. Croisier, M. Israel, F. Chavand, Eds., IEEE Computer Society Press: Los Alamitors, CA, 1993.
5. E. W. Dijkstra, "The structure of the 'THE'-multiprogramming system." *Comm of ACM* 11(5), pp. 341–346, May 1968.
6. C. Hewitt, "Viewing control structures as patterns of passing messages." *Artificial Intelligence* 8, pp. 323–364, 1977.
7. S. Ginali, J. Goguen, "A categorical approach to general systems." *Applied General Systems Research,*G. J. Klir, Ed., Plenum Press: New York, 1978.
8. D. H. H. Yoon, "The categorical framework of object-oriented concurrent systems." *Computers and Mathematics with Applications* 25(2), pp. 33–38, 1993.
9. B. H. Liskov, "The design of the Venus operating system." *Comm of ACM* 15(3), pp. 144–149, March 1972,
10. E. I. Organick, *The Multics System: An Examination of Its Structure.* MIT press: Cambridge, MA, 1981.
11. P. K. Andleigh, *UNIX System Architecture.* Prentice-Hall: Englewood Cliffs, NJ, 1990.
12. J. D. Day, H. Zimmermann, "The OSI reference model." *Proc. of the IEEE* 71(12), pp. 1334–1340, Dec. 1983.
13. O. J. Dahl, K. Nygaard, "SIMULA—An algol-based simulation language." *Comm of ACM* 9(9), pp. 671–678, 1966.
14. G. Booch, *Object Oriented Design with Applications.* Benjamin/Cummings: Reading, MA, 1991.
15. G. Birkhoff, J. D. Lipson, "Heterogeneous algebras." *J. of Combinatorial Theory* 8, pp. 115-133, 1970.
16. J. A. Goguen, J. W. Thatcher, E. G. Wagner, "An initial algebra approach to the specification, correctedness, and implementation of abstract data types." T. Yeh, Ed., *Current Trends in Programming Methodology.* IV: Prentice-Hall, 1978, pp. 80–149.
17. D. C. Parnas, "A technique for software module specification with examples." *Comm of ACM* 15(5), pp. 330–336, 1972.
18. D. C. Parnas, "On the criteria to be used in decomposing systems into modules." *Comm of ACM* 15(12) pp. 1053–1058, 1972.
19. H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 2.* Springer-Verlag: New York, 1990.
20. K. Futatsugi, J. A. Goguen, J. P. Jouannaud, J. Meseguer, "Principles of OBJ2." *Proc. 12th Annual ACM Symposium on Principles of Programming Languages.*
21. B. Stroustrup, *The C^{++} Programming Language.* 2nd ed, Addison-Wesley: New York, 1991.
22. D. A. Marca, C. L. McGowan, D. T. Ross, *SADT.* McGraw-Hill: St. Louis, 1988.
23. G. Bate, "MASCOT 3: An informal introductory tutorial." *Software Eng. J.* 1(2), 1986.

24. B. Delatte, M. Heitz, J. F. Muller, *HOOD:Reference Manual 3.1.* Prentice-Hall: London, 1993.
25. D. G. Firesmith, *Object-Oriented Requirements Analysis and Logical Design.* John Wiley & Sons: New York, 1992.
26. W. Reisig, *A Primer in Petri Net Design.* Springer-Verlag: New York, 1993.
27. D. Harel, "On visual formalisms." *Comm of ACM* 31(5), pp. 171–187, May 1988.
28. International Standards Organization, "Information processing system-open systems inter-connect- basic reference model." ISO 7498, International Standards Organization, Geneva, 1983.
29. M. Maekawa, A. E. Oldehoeft, R. R. Oldehoeft, *Operating Systems: Advanced Concepts.* Benjamin/Cummings, Pub. Co: Menlo Park, CA, 1987.
30. L. S. King, "Hierarchical CIM lab control." *Proc. ASEE 1994 North Central Section Spring Conference.* Grand Rapids, MI, April 7–9, 1994.
31. M. Brill, U. Gramm, "MMS: MAP application services for the manufacturing industry." *Computer Networks and ISDN Systems* 21, pp. 357–380, 1991.