Computers Math. Applic. Vol. 25, No. 2, pp. 33–38, 1993
Printed in Great Britain. All rights reserved
0097-4943/93 $5.00 + 0.00
Copyright© 1992 Pergamon Press Ltd

THE CATEGORICAL FRAMEWORK OF OBJECT-ORIENTED CONCURRENT SYSTEMS

D. H. H. Yoon
Department of Computer and Information Science, University of Michigan-Dearborn
Dearborn, MI 48128, U.S.A.

(Received September 1991 and in revised form March 1992)

Abstract—Category Theory is introduced as a mathematical model for an object-oriented concurrent system which is viewed as a collection of objects and processes. An object can be represented as an algebra, whereas a process as a subalgebra.

1. INTRODUCTION

One of the major challenges facing today's computer scientists is developing a computer system for an autonomous mobile robot which constantly interacts with its environment, updates its knowledge base including the world model, and carries out tasks specified by humans. Due to the massive influx of input data, the system has to perform computations concurrently. Also, human users should be able to interact with the system in the object-oriented fashion. Such a system can be described as an object-oriented concurrent system (OOCS) which can accommodate new components. The most fundamental elements of such a system are objects and processes: An object is a human-oriented concept in that humans think of the real world in terms of objects, whereas a process is machine-oriented. A category is introduced as a mathematical model for the system, and objects and processes are precisely defined in the categorical framework.

2. THE THEORETICAL MODEL FOR AN OBJECT-ORIENTED CONCURRENT SYSTEM

Just like other computer systems, the object-oriented concurrent system consists of users, software and hardware. Considering users as a system component is one of the most important assumptions to be made in developing an object-oriented system because the final product should be user-friendly and easy to use. In order to achieve these goals, the system should be designed in such a way that users can communicate with it in terms of objects.

The objects in a user's program will be eventually partitioned into processes inside the system. However, the users do not have to know the presence of processes in the system. The better shielded the processes are from the users, the better the system will be.

Software will be viewed as the collection of autonomous processes which communicate with each other: A process does not know the inner workings of other processes, but interacts with them through communication channels. The mathematical model based on these assumptions will become the foundation for developing both software and hardware for object-oriented systems and is a variation of a category.

Definition 1 [1]. *A category C consists of*

(i) *a class of elements O called objects of C,*
(ii) *for each object $A \in O$ a set $U(A)$, called the universal set of A,*
(iii) *for each pair of objects, A, B, a set* $\hom(A,B)$ *of maps from $U(A)$ to $U(B)$, called the morphisms from A to B in C, these being subject to the following conditions:*
 (a) *for each object A,* $\hom(A,A)$ *contains the identity $1_{U(A)}$;*
 (b) *for objects A, B, D from C, when f, g are from* $\hom(A,B)$ *and* $\hom(B,D)$, *respectively, the composite map $g * f$ belongs to* $\hom(A,D)$.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

The identity map mentioned in (iii, a) of the above definition is always suppressed in a computer system.

A category is an abstraction of sets and functions defined on them. Using this fact, we formulate an object-oriented concurrent system as follows:

$$\mathrm{OOCS} = \langle O, S \rangle,$$

where O is the collection of objects and S is system software. The objects are the internal representations of entities such as physical objects, abstract data types and robot motions and constitute the user space. The question of object-orientation arises in this space. On the other hand, system software S consists of processes, and concurrency is the problem due to the interactions between processes. An object has to be eventually blended into the world of processes, which we call the kernel space. In the next two sections, an object and a process are precisely defined using the notion of algebra and their relationship will become clear as the paper progresses.

2.i. The Space of Objects O

As mentioned earlier, a computer system consists of users, software and hardware. In an operating system like UNIX the three spaces are clearly defined. Objects are the fundamental elements of the user space, whereas processes are those of the kernel space realized by system software. An object, in this paper, refers to an entity that a user defines and is fully self-contained except communication with other objects. A process, on the other hand, is created by an operating system and shares similarities with objects.

Some common characteristics of both object and processes are that they are allowed to access only their own respective local variables and that communication is achieved through legitimate means. An object defined in the user space will be partitioned into processes as the object is brought into the kernel space. How to partition an object into processes depends on the swapping and paging policies of the system and we will not consider the details. Instead, the common structure of both object and process will be discussed in detail using the notion of an algebra.

2.i.a. Algebras

We have defined OOCS as the collection of objects and software using the concept of category. It turns out that the mathematical model for an object is an algebra and that a process can be represented as a subalgebra. In order to introduce the concept of algebra, we start with an example which most computer scientists feel comfortable with. The ADT (abstract data type) stack is specified in Figure 1 [2].

Using the ADT stack as an example, we now introduce the notions of sort, signature, algebra and initial algebra. A sort corresponds to the collection of data types employed in the ADT stack. A signature introduces the names of constants and functions as in the operation section in Figure 1. Associated with each function name, the arity of the function is the number of arguments of the function. For example, arity(**zero**) = 0, arity(**succ**) = 1 and arity(**push**) = 2. Now let us define an algebra.

DEFINITION 2. *If Σ is a signature, a Σ-algebra is a pair $\langle A, \Sigma_A \rangle$ where*

(i) *A is a set, called the sort,*
(ii) *Σ_A is a set of functions $\{f_A : f \in \Sigma\}$, such that if* arity*$(f) = n$, then*

$$f_A : A^n \to A.$$

Relating the definition to the stack example, the sort A = {**stack**, **nat**, **bool**}, the signature Σ = {**true**, **false**, **zero**, **succ**, **newstack**, **push**, **isnewstack**, **pop**, **top**} and Σ_A consists of all the entries in the axiom section. The signature essentially specifies the syntax of the functions, whereas its algebra specifies the semantics.

As illustrated above, an object is represented as an algebra. Hence, a category can be rephrased as a collection of Σ-algebras. Now we will define an object in the categorical framework.

```
TYPE  stack_type;
SORTS stack; nat; bool;
OPERATIONS
  true, false:-> bool;
  zero:-> nat;
  succ: nat -> nat;
  newstack:-> stack;
  push: stack*stack -> stack;
  isnewstack: stack -> bool;
  pop: stack -> stack;
  top : stack -> nat;
declare s: stack; n:nat;
  AXIOMS
  isnewstack (newstack) == true;
  isnewstack (push(s,n)) == false;
  pop (newstack) == newstack;
  pop ( push(s,n)) == s;
  top (newstack) == zero;
  top (push(s,n)) == n;
  ENDTYPE.
```

Figure 1. ADT stack.

2.i.b. *An object*

Now let us view a category C as a collection of Σ-algebras and let $\langle A, \Sigma_A \rangle$ and $\langle B, \Sigma_B \rangle$ be two Σ-algebras in C. A function $h : A \to B$ is a Σ-homomorphism if for every $f \in \Sigma$ of arity k

$$h(f_A(a_1, \ldots, a_k)) = f_B(h(a_1), \ldots, h(a_k)).$$

A Σ-homomorphism $f : A \to B$ is called a Σ-isomorphism if it is one-to-one and onto. In order to define an object in the categorical sense, we need the notion of initial algebra.

DEFINITION 3. *Let C be a category of Σ-algebras. Then, a Σ-algebra $I \in C$ is initial in C if for every Σ-algebra $J \in C$, there exists a unique Σ-homomorphism from I to J.*

Now we are ready to define an object formally.

DEFINITION 4. *An object is the isomorphism class of an initial algebra in a category of Σ-algebras.*

Let us elaborate the above definition in terms of the ADT stack. An empty stack created by the operation `newstack()` is regarded as an initial algebra. Different instances of the stack due to a finite number of `push()` and `pop()` operations on it belong to the isomorphism class of the stack, and, hence, the ADT stack is an object.

Hence, the set O in the definition of an object-oriented concurrent system can be viewed as a collection of initial algebras of a category. In the next section, we define a process in terms of a subalgebra.

2.ii. *System Software S*

The system software S can be viewed as the collection of processes acting on objects in the system. The notion of process is so crucial in developing system software that we review how the term process has been used in traditional operating systems and define it algebraically. A process is a computing agent consisting of instructions, local data and a stack. The program segment consisting of instructions acts on the data. The stack is created by an operating system to keep track of whereabouts of the process within the system. Hence, a process can be regarded

as a collection of a program segment and local data including the stack. It is self-contained and communicates with other processes through a communication mechanism.

The relationship between an object and processes is very important. In the previous section, we introduced an algebra as the mathematical model for an object. The object, when introduced into the kernel space, is partitioned into one or more processes by the operating system. For instance, a module implementing one of the functions in the stack example could be a process to the operating system. Since an object has been represented as an algebra, a process will be represented as a subalgebra to emphasize the relationship between an object and processes.

DEFINITION 5. *A process is the isomorphism class of an initial subalgebra in an algebra.*

Throughout this paper we have considered only user processes. However, a process can execute in either user or kernel mode. The process executing in kernel mode is known as a system process. For example, a bootstrap module which boots a system is an important system process. Hence, system software consists of user and system processes.

2.iii. Communication

A process has been informally described as a computing agent which is capable of communicating with other processes. Communication between processes is one of the most important tasks in an object-oriented concurrent system. For example, communication between sensors and a robot or between actuators and a robot remains to be resolved. This problem boils down to communication between processes. In this section, we consider two important communication schemes: classical and message-passing.

The classical scheme is based on the input/output operations of a computer system. In this scheme, participating processes are equipped with ports through which communication takes place. Since a process is a computing agent, it can be described as a formal machine. Following Steenstrup *et al.* [3], we describe the sending and receiving operations of a process as a port automaton.

DEFINITION 6. *A port automaton P is a collection of objects and maps $(L, Q, \tau, \delta, \beta, X, Y)$, where L is the set of ports, Q is the set of states, $\tau \in 2^Q$ is the set of initial states, $X = \{X_i : i \in L\}$, where X_i is the input set for port i, $Y = \{Y_i : i \in L\}$, where Y_i is the output set for port i, $\delta : Q \times \cup X_i \to 2^Q$ is the transition map, $\beta = \{\beta_i : i \in L\}$, where $\beta_i : Q \to Y_i$ is the output map for port i, all subject to the axiom that for $q \in Q$, $\{x \in X : \delta(q,(x,i)) = \emptyset \text{ or } X_i\}$.*

It is a common belief that this technique alone might be too application-dependent and may not be suitable for a concurrent system, which should be able to add new components without damaging the integrity of the system. However, when employed along with the message-passing technique, this technique appears to be essential in a communication system.

Compared to the classical technique, the message-passing technique is flexible and has been successful in many applications. There are two modes of passing messages in use: direct and indirect. In the direct system, a message is directly sent to a process, say P, or received from another process, say Q. Two primitive operations of this system are sending and receiving as follows [4]:

send (P, message) = send a message to process P,

receive (Q, message) = receive a message from process Q.

The disadvantage of this mode is, when a process has a new name in a system, it is necessary to examine all the process definitions in the system. This situation is not desirable. To overcome this drawback, indirect mode is often used.

In the indirect system, a message is sent from a mailbox and received by a mailbox:

send (A, message) = send a message to mailbox A,

receive (B, message) = receive a message from mailbox B.

There are various issues in the indirect message system such as the ownership of a mailbox. A detailed discussion of the topic is beyond the scope of this paper and we conclude this section by reiterating that communication is one of the central issues in a concurrent system.

2.iv. An Illustration

As an application of the theory developed so far, we present a brief description of a workcell in a computer integrated manufacturing (CIM) system in which computers play the major role. The workcell (Figure 2) described in this section is very similar to the one in [5]. It consists of a workcell controller, two workstations, and a transport system. Roughly speaking, the workstation A produces products and places them on the transport system T. Then the workstation B processes and outputs them to the environment. The workcell controller is responsible for the operations of the workcell and communication between the workcell and the environment.

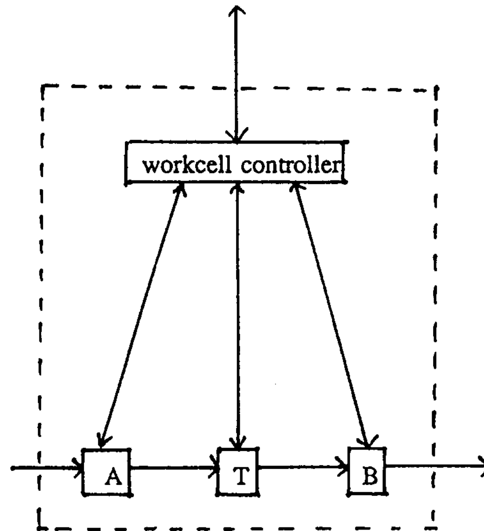

Figure 2. A simple workcell.

To be more specific, a 'workstation' is the smallest unit of the manufacturing system that can be commanded to test, store, and transform a product. A 'transport system' is a unit that can be commanded to accept products from senders and to transport them to receivers. A 'workcell controller' is a system that interfaces with the environment, and tells workstations which operations to execute and the transport system where products must be transported to.

The above workcell, as a whole, can be modeled as a process which communicates with its environment. The process then is partitioned into subprocesses which are further refined. Hence, the entire workcell is considered as a hierarchy of process definitions.

A complete specification of the workcell is beyond the scope of this paper. However, we illustrate how the theory developed in the previous sections can be employed in the specification of the workcell by describing two primitives of the workcell, the product type and the command type in terms of algebras.

Let us assume that there are two kinds of products manufactured in the workcell, **product_1** and **product_2**, and further, that **product_2** can be obtained by processing **product_1**. The **product_type** can be specified as follows:

```
TYPE    product_type;
SORTS   product;
OPERATIONS
   processed : product --> product;

declare product_1, product_2: product;

AXIOMS
   processed (product_1) == product_2;
   processed (product_2) == product_2;
 ENDTYPE.
```

The other primitive of the workcell is concerned with communication between the workcell controller and workstations. As mentioned earlier in this paper, a process is an agent communicating with other processes, and hence, the means of communication is one of the most important aspects of OOCS. The workcell controller communicates with a workstation by sending a command to and receiving a status report from the workstation. Both a command and a status report can be specified in terms of algebra as follows:

```
TYPE command_and_status_type;
SORTS operation_command, ready_status;
OPERATIONS
  command: integer --> operation_command;
  number_in: operation_command--> integer;
  ready :  --> ready_status;
 AXIOM number_in(command (n)) == n;
ENDTYPE.
```

Here, we have presented specifications of two primitive elements of the CIM system for the illustrative purpose. However, the specification of the entire workcell requires additional primitive processes. Then each component of the workcell such as the controller and workstations will be specified in terms of the processes which have been previously defined.

Such a formal specification is invaluable in the development of computer systems at least for two reasons: First, a good formal specification method eliminates ambiguities in the description of a system. Second, it allows a designer to test the system prior to its implementation so that he can avoid specifying a system which will fail later.

3. CONCLUSION

An OOCS, which supports object-orientation, concurrency and dynamic adaptation, is a very complex system. In order to develop such a system, a firm theoretical foundation is urgently needed. A few attempts have been made to develop such systems. However, it appears that further research needs to be carried out at the theoretical level. Toward this effort, we have presented the categorical framework for an object-oriented concurrent system in which objects and processes are the fundamental elements.

REFERENCES

1. V.S. Krishnan, *An Introduction to Category Theory*, North-Holland, New York, (1981).
2. I.V. Horebeek and J. Lewi, *Algebraic Specification in Software Engineering: An Introduction*, Springer-Verlag, New York, (1989).
3. M. Steenstrup, M.A. Arbib and E.G. Manes, Port automata and the algebra of concurrent process, *J. Computer & System Sciences* **27** (1), 29–50 (1983).
4. J.L. Peterson and A. Silberschatz, *Operating Systems Concepts*, 2nd ed., Addison-Wesley, Reading, MA, (1985).
5. F. Biemans and P. Blonk, On the formal specification and verification of CIM architecture using Lotos, *Computers in Industry* **7** (6), 491–504 (1986).