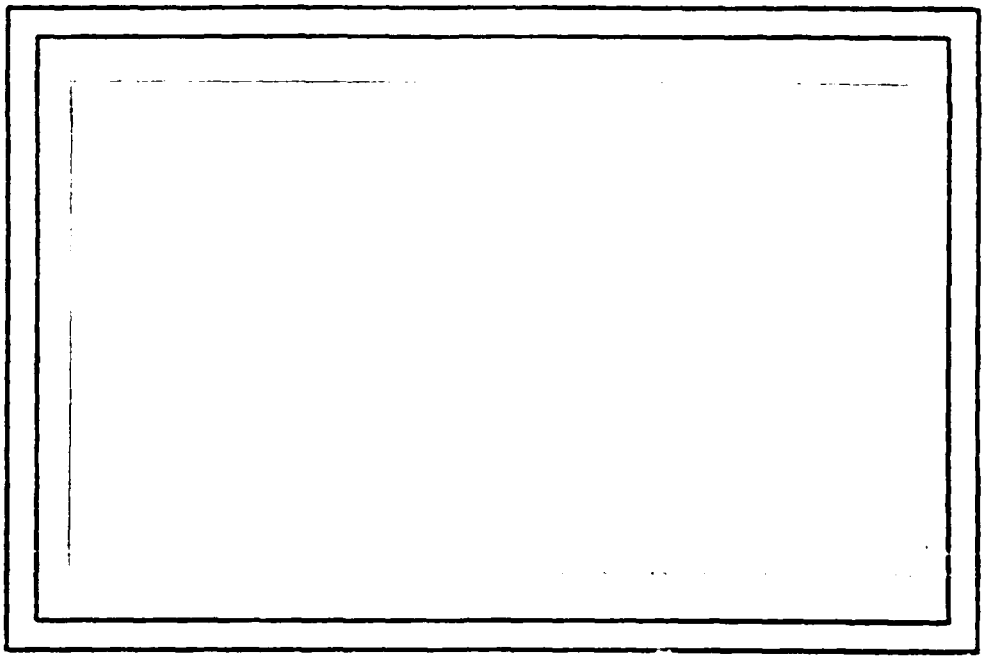
(4)

DTIC FILE COPY

AD-A199 009



COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

4

CS-TR-2003 March, 1988

An Object Architecture for Hard Real Time Systems *

Juergen Nehmer **

Systems Design and Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

The paper deals primarily with the question to what extent the object model proposed as a powerful structuring method for large software systems could serve as a suitable basis for the construction of hard distributed real time systems. A particular object model is proposed which supports the idea of object autonomy in a perfect way. It is shown that object autonomy is a key factor to meet extreme real time and fault tolerance requirments.

* This work is supported in part by contract DASG60-87-C-0066 from the U.S. Army Strategic Defense Command and by contract N00014-87-K-0241 from the Office of Naval Research to the Department of of Computer Science, University of Maryland.

** The author was on leave from Universitaet Kaiserslautern, D6750 Kaiserslautern, Fachbereich Informatik, Postfach 3049, West Germany

1. INTRODUCTION

The present generation of real time operating systems supports real time requirements in a very indirect way: designers of real time systems are merely provided with some handles to control scheduling disciplines in the underlying operating system thereby minimizing timing uncertainties. Examples of those control mechanisms are OS functions for the assignment and change of process priorities or for locking pages in main memory in order to avoid unexpected delays of a program execution due to paging[27,28]. This philosophy in present day real time systems leaves the responsibility of assuring that real time constraints are met up to the user. Application designers have to define the desired process priorities, pages to be locked etc. in an off-line procedure which includes also a validation step for checking that the real time requirements are met. No actual checking of real time constraints is performed by the real time system at run time. A small change of the system configuration(hard-or software) raises the need to stop the system and restart it later on after the successful off-line validation step.

The future generation of hard real time operating systems as for example MARUTI[1,3] claim to meet real time constraints with a specified degree of reliability on an interactive request: the real time conditions of a given application are checked against the present system load and granted to the user by an internal reservation policy. This yields the following important consequences:

- The responsibility for assuring that real time constraints are met is shifted from the user to the operating system: it is up to the operating system now to assign proper priorities to processes, lock pages in main memory etc. The corresponding mechanisms are not accessible to users any more.

-Real time operating systems with this capability support an open job shop: the system load can vary dynamically due to new applications or modifications of the hardware configuration as long

as the operating system has granted to continue meeting the actual real time constraints.

These properties of future hard real time operating systems pose new challenges on adequate means and concepts for their construction.

This paper deals with the question to what extent the object model proposed as a powerful structuring means for many kinds of systems[26] could also serve as a suitable basis for the construction of hard real time systems. It is argued that an object architecture which enables designers to decompose their system into a set of largely autonomous units seems to meet realtime requirements best.

In chapter 2 we will give a more precise definition of what is understood by functional autonomy and discuss its role as a structuring principle in realtime systems.

In chapter 3 the elements of the proposed object model are introduced.

The power of the object model for the description of typical realtime tasks is demonstrated by three examples in chapter 4.

Basic implementation considerations for the object model are presented in chapter 5.

In chapter 6 the issue of fault tolerance and its relation to an adequate object architecture is raised. These thoughts are of preliminary nature and need further investigations.

2. THE PRINCIPLE OF FUNCTIONAL AUTONOMY IN REALTIME CONTROL PROGRAMS

Realtime control programs are usually composed from a set of function modules which form a hierarchical control structure as depicted in Figure 1. A module at a higher level controls adjacent lower level modules by coordinating their activities and /or providing directives for their particular tasks.

In well structured realtime programs this decomposition in control layers is done in such a way that lower level modules adhere to the principle of (limited) functional autonomy. This principle

is established for a given module by three properties:

1. The module is in possession of all resources to perform its function when requested. If resources are shared with other modules a transparent resource allocation mechanism assures proper allocation of missing resources in time whenever they are needed. At its extreme resources are allocated permanently to a module. Distributed systems are particularly suited to support this philosophy.

2. Once created a module may initiate future activities in its own rights, i.e. it doesn't need external stimulations by other modules. This rules out mechanisms such as the remote procedure call as the only means for the activation of operations within a module.

3. Once an operation has been initiated within a module it can run through completion without invocation of external services provided by other modules.

Strict observance of this principle yields the following noticeable advantages:

-A low level module can continue operation for a certain period of time in the absence of its supervisory control module.

- Fault tolerance may be established on the module basis with regard to the special reliability degrees required in each module. A problem might arise in systems in which some modules are shared between applications with different reliability requirements. This problem will be addressed in a broader context in a later chapter.

- Critical realtime constraints within a module can be treated locally. They have no effects on modules having their own resources associated to them. For example, a module containing an operation which needs periodic activation every 5 msec might require a high speed processor. All other modules may still perform their function within their specified time limits using a low speed processor.

It can be observed that proper application of the principle of functional autonomy results in

hierarchical system structures with the hardest realtime and fault tolerance requirements located at the bottom layer. With increasing distance from the bottom layer realtime and reliability constraints decrease while the need for more computing power and storage capacity increases.

3. AN OBJECT MODEL SUPPORTING FUNCTIONAL AUTONOMY

In this chapter a simple object model is introduced which supports the idea of functional autonomy as introduced in the previous chapter. We also present an elegant way to express realtime constraints. The model is described in a rather abstract way: we look at objects from a language point of view and postpone implementation issues to a later chapter.

Our proposal for an object model is closely related to Hewitt's actor model[4,5,12]: objects can be thought of consisting of a collection of operations which manage an internal state. An object type can be described as:

```
OBJECT ObjTypeName
  Declaration of internal state;
OPERATION OperationName1(--Parameters--)
  DO statement sequence END
OPERATION OperationName2(--Parameters--)
  DO statement sequence END
  .
  .
  .
BEGIN
  initialization of the object's state;
END ObjTypeName
```

Several object incarnations(called objects) may exist from an object type at any time:

```
a,b,c:ObjTypeName;
```

In the above example three objects named a,b,c have been created by a declarative statement. It

is assumed that the initialization part of each object gets executed after object creation.

Another way of object creation is by calling a CREATE operation of the runtime environment or the operating system.

An object gets activated within an object after arrival of a message which has been send to this operation. The basic message handling semantics is assumed to be asynchronous, one-way: a sending object is blocked until the message has been passed successfully to a transport system. It is assumed that the transport system will deliver the message in an undefined but limited time to its destination. The arrival at the specified destination triggers the activation of the corresponding operation within the addressed object. The successful execution of an object's operation is called an action.

This simple object model described above served as the basis for the distributed object oriented languages CSSA[9,10,18] and POOL[6].

Since we assume implicit message receipt by an arriving message and since no replies are defined in a one-way message handling scheme the only visible message handling primitive in the operations part of objects is a SEND primitive.

We propose that the SEND primitive takes the following form:

SEND <message> TO <destination> AT <time constraints>;

The <message> part may include additional control information such as semantic link information in joints[2].

The <destination> part may be

a. a constant specifying an operation within an object by a qualified name "ObjectName.OperationName"

or

b. a constant specifying an operation within an object type by a qualified name "ObjectType.OperationName"

or

c. a variable of type operation name containing a value of either (a) or (b) above.

If an object type was specified in the destination part it is assumed that the message will be sent to all objects which presently exist of this type. This will provide for a multicast capability of the proposed message handling scheme which is considered particularly useful for the implementation of fault tolerant structures. We will postpone this discussion to a later section.

The AT-part of the SEND operation allows the explicit definition of timing constraints connected with the action specified in the SEND statement. It is proposed to provide expressions which describe relative time intervals or the absolute time at which the addressed operation should be carried out.

With the above explanations an operation takes the form:

```
OPERATION OperationName(--Parameters--)
 DO
  .
  .
 SEND...
 SEND...
  .
  .
  .
END OperationName
```

Since the statement sequence does not contain statements with non- deterministic timing behavior(due to the principle of function autonomy and to the semantics of the SEND operation) every statement is time bound and operations are also time bound as a whole therefore.

The property of time bounded operations together with the explicit specification of time constraints for actions in the AT-part of the SEND primitive forms a sufficient basis for the development of smart schedulers which are capable to check and guarantee realtime constraints[2].

If a message arrives at an object the addressed operation is assumed to execute atomically with respect to simultaneous message arrivals at the same object(the term atomicity is used here with respect to the indivisibility of the operation, a more comprehensive definition is given in [22]).

The most rigorous way of assuring atomicity is to enforce that operations within an object exclude each other in time as in CSSA[9]. A more elaborate way which avoids unnecessary waiting delays in case that actions within one object are disjunct with respect to the data they access is to principally allow parallel actions within an object and preserve atomicity by specifying mutual exclusion relations between them[2].

4. EXAMPLES

In order to demonstrate the potential power of the proposed object model for the description of typical realtime control structures three examples are discussed below. The examples also illustrate good and bad usage of the autonomy principle.

Example 1: Action sequence triggered by an interrupt

It is assumed that a sequence of actions in n different objects Obj1..ObjN is triggered by an interrupt signal. The interrupt signal is treated as a message sent by a predefined system object. The object interconnection structure is shown in Figure 2a.

The first n-1 objects contain only one operation with the following structure:

```
OBJECT ObjX
OPERATION Op(--Parameters--)
 DO
 .
 .
 .
 SEND Message TO ObjX+1.Op  AT(0);
 END Op;
BEGIN
 Initialization;
END ObjX
```

The last object in the sequence does not contain any SEND statement. Figure 2b shows the action sequence in a time diagram.

In this example an action in an object is triggered by a SEND primitive within another object, i.e. in order to perform some function an object relies completely on external stimulations. Since the autonomy criteria 2 above is violated the autonomy of all objects is very poor

Example 2:Sequencing of actions by a supervisory object

The almost same effect as in example 1 can be obtained by a supervisory object which enforces sequential execution of actions represented by the objects Obj1 .ObjN

The object interconnection structure is shown in Figure 2a The supervisory object reveals the following basic structure:

```
OBJECT Supervisor
VAR I: Integer

OPERATION Op(--Parameters--)
 DO
  IF I<N THEN
        I:= I+1;
        SEND Message TO Obj1.Op  AT(0);
       ELSE  I:=0;
 END;
 END Op;
BEGIN
 I:=0;
END Supervisor
```

The objects Obj1..ObjN follow the basic pattern as depicted below:

```
OBJECT ObjX
 OPERATION Op(--Parameters--)
  DO
  .
  .
  .
  SEND Message TO  Supervisor.Op  AT(0).
  END Op;
BEGIN
Initialization;
END ObjX
```

It is assumed that an interrupt message from a predefined system object will be sent periodically to the operation Op of object Supervisor. As in the previous example all actions in the objects Obj1..ObjN are triggered by external messages, i.e. they are completely dependent of the Supervisor object in order to perform their functions. Therefore, the object autonomy is again very poor since the autonomy criteria 2 has been violated. Figure 3b shows the resulting action sequence in a time diagram.

Example 3: Autonomous objects

The last example represents a system in which the principle of autonomy has been realized in an ideal way. There, a set of objects Obj1..ObjN perform their function largely independent from a supervisory object. Occasionally the object Supervisor generates a set of updated parameters for the objects Obj1..ObjN thereby enabling them to perform their task in a more optimal fashion However, if the object Supervisor fails it is assumed that all objects Obj1..ObjN may continue operation under less optimal conditions.

The object interconnection structure for this example is shown in Figure 4a.

The Supervisor object reveals the following basic structure:

```
OBJECT Supervisor
 OPERATION Op(--Parameters--)
  DO
  Prepare parameters for objects Obj1..ObjN;
  FOR I=1 TO N
   DO
   SEND Message1  TO ObjI Op  AT(0);
   END;
  SEND Message2  TO Supervisor.Op  AT(ΔT1);

  .

 END Op;
BEGIN
 Initialization.
END Supervisor
```

The objects *Obj1..ObjN* export two operations *Op1* and *Op2*. The operation *Op1* accepts the initial parameter set and stores them internally for future use. It then issues the first SEND to its own operation Op2. If Op2 has been activated once it will continue activating itself periodically by an appropriate SEND statement.

This leads to the following basic structure for the objects Obj1..ObjN:

```
OBJECT ObjX
 OPERATION Op1(--parameters--)
  DO
  Store parameters;
  SEND Message2  TO ObjX.Op2  AT (0);
  END Op1;

 OPERATION Op2(--parameters--)
  DO
  SEND Message2  TO ObjX.Op2  AT(ΔT2);
  .
  .
  .
  END Op2;
BEGIN
 Initialization;
END ObjX
```

It should be finally mentioned that inherent interrelations between different functions of a real-time application might make it very hard to structure a system according to the principle of functional autonomy. However, if a certain degree of functional autonomy has been achieved it should be completely retained in the final realization by corresponding autonomous objects.

5. BASIC IMPLEMENTATION CONSIDERATIONS

The following basic implementation considerations ignore specific reliability requirements for the construction of fault tolerant structures. Those questions are addressed in the next chapter.

Figure 5 reflects the basic implementation idea for objects proposed here. An object is considered to constitute a team[11,18] of a dynamically changing number of light-weight processes.

After object creation an object consists of exactly one process-the root process-and an empty message queue. The root process will exist as long as the object exist. Its mere purpose is to listen to the message queue for incoming messages. When a new message has arrived a pointer to it is passed to the root process and the message is marked as "recognized". The root process creates a

light-weight process of type "server" and passes the pointer of the recognized message to it. It then forgets the message and waits for the next message to arrive.

This leads to the following basic implementation structure of the root process:

```
PROCESS Root
DO
 LOOP
 MessagePtr=WAIT;
 ProcId=CREATE_PROCESS(MessagePtr);
 END;
END Root
```

Each server process once created reads the message from the message buffer which turns the message into the state "copied". The server process then identifies the operation which should get executed and calls a subroutine to do the job(it is assumed that a reentrant procedure is provided for each operation exported by an object). After successful completion of the call the server process removes the message from the message buffer. This leads to the following basic implementation structure for a server process:

```
PROCESS Server(MessagePtr)
DO
READ(MessagePtr,Destination);
CALL OperationName(--Parameters--);
REMOVE(MessagePtr);
END Server
```

The proposed message handling scheme has the effect that every existing message in a message buffer will be in one of three states:

-untouched
-recognized
-copied

An untouched message has not been recognized yet by the object. A recognized message has been

taken into account by the object but no action has been carried out so far by the object. A copied message has been successfully stored in the local address space of the object. The requested operation

might not have been started yet, might be in progress or was completed already. For a removed message the corresponding action has been carried out successfully.

The proposed scheme of late message removal in combination with the additional message state information may serve as a basis for efficient recovery procedures in case of node crashes.

Assume for example that the following messages with the indicated states are hold in a message buffer shortly before the node at which the corresponding object is located crashed(it is assumed that the message buffer uses a physically separate storage which survived the crash):

-message M1:copied
-message M2:recognized
-message M3:untouched

For an object with idempotent operations[11] the object can simply be recovered by turning the states of messages M1 and M2 to "untouched" and restart the root process again.

If the operations not idempotent only the state of M1 is turned into "untouched". The message M3 is either removed or turned into the state "untouched" after a clean-up of the object's internal state depending on whether a forward or backward error recovery policy is in effect[7,19].

Notice that the proposed implementation scheme may lead to an arbitrary number of simultaneous server processes existing within an object. However, since the server processes all share the common working store of their object and since they are of the identical process type(which is assumed to be a code template residing permanently in the object's working store) efficient light weight implementations are possible.

Another problem connected to simultaneous server processes within an object is the need for synchronizing the accesses to shared data. We assume that the formulation of mutual exclusion

conditions for operations as part of the object's description will enable a clever compiler to insert semaphore operations at the appropriate places so that we can further ignore this problem here.

Applying the proposed object architecture for structuring complete systems results in distributed programs[]. With regard to the criteria

retaining functional autonomy it is highly recommended to locate objects on separate processing nodes. This leads naturally to a distributed systems approach for the realtime systems under consideration.

6. PRELIMINARY THOUGHTS ON FAULT TOLERANCE

The potential of implementing any desired degree of fault tolerance is still the most crucial(and most challenging) requirement for realtime systems.

In this section we give a brief overview of current systematic approaches to fault tolerance with respect to distributed system environments. We then discuss the suitability of these methods for realtime systems. The discussion is focused on the most popular class of failures, the so called fail-and stop failures[7]. A typical example for this class of failures is a processor crash due to disruption of power supply. There are other classes of failures which don't show the nice behavior of fail-and stop failures which need a more sophisticated treatment.

If a failure occurred we assume that an operation couldn't run through completion thus leaving the object in an inconsistent state(this is the most general case).

All published schemes for handling such a situation are based on the restoration of a consistent state where a restart is possible[7,19,20]. Notice that a local recovery step within the object is generally insufficient since the locally restored state might conflict with global conditions in other objects. This raises the need for a global recovery procedure after each single object failure. A consistent state after a failure could be obtained in two different ways:

-In a backward error recovery scheme the system state is reset to some saved consistent state passed through in the past after replacement of the failed component.

-In a forward error recovery scheme the history is basically lost and the system state is set forward to some well known consistent state for a later restart.

Backward and forward error recovery schemes differ substantially in the basic organization and algorithms applied to reach the goal. They are also heavily dependent on the basic architecture of the software.

If an object architecture with a synchronous RPC-like interaction mechanism between objects is used (called the object/action paradigm in [22]) the atomicity property for actions as defined in [14] will provide for an adequate backward error recovery scheme.

If a software architecture based on asynchronous message passing is used (referred to as the process/message paradigm in[23]) checkpointing in combination with automatic rollback has been proposed as the appropriate backward error recovery scheme[]. A particular problem of recovery in asynchronous systems is the potential of rolling back to the initial system state called the "domino effect" which has been analyzed by several authors[13,21].

Randell[23] has shown in a recent paper that atomic actions and checkpointing are dual recovery concepts with basically identical power.

Unfortunately, not too many results of the outlined recovery schemes proposed so far are applicable in the realtime world for the following reasons:

1. Backward error recovery is generally not suited for realtime applications since the environment controlled by the computer moves forward to a state which affects the restored system state. Therefore forward error recovery is the more realistic technique to handle failure situations in realtime systems.

2. The implications of forward error recovery in a distributed realtime environment don't seem to be well understood at the moment and only a few papers addres. the problem in a more general sense[8,25].

3. Forward error recovery cannot make effective use of history information gathered by some state saving mechanism. Thus neither checkpointing nor atomic actions seem to be the schemes on which fault tolerance considerations may be based on.

The above discussion can be summerized as follows:

1. Fault tolerance schemes for realtime systems should be based on forward error recovery.

2. More basic research investigations are necessary in order to understand the nature of forward error recovery in the context of distributed realtime systems and to come up with a general scheme for handling failure situations.

Let us investigate now the suitability of the proposed object model for the realization of fault tolerant systems as far as this seems to be possible right now.

The first question to be answered is: Given a system of interacting objects, should fault tolerance be associated

with objects or with actions?

Since objects could be shared by different applications with differing fault tolerance degrees associating fault tolerance with actions would raise the need for dynamically changing the fault tolerance degree of objects. It is our opinion that efficiency reasons rule out such a strategy. Therefore it is assumed that fault tolerance is a property which is always connected to objects. If objects are shared between applications with different fault tolerance requirements it is assumed that the highest degree of fault tolerance requested for a particular object will be installed.

We propose that for any particular object one may choose a specific degree of fault tolerance from

a predefined list known to the system. Below is a list with possible fault tolerance degrees:

(a) none
(b) stable storage[17]
(c) atomicity of actions
(d) passive stand-by redundancy
(e) active stand-by redundancy
(f) 3-modular redundancy

A single failure will leave an object with fault tolerance degree

(a) in an undefined state,

(b) in an undefined state but with crucial information saved on stable storage,

(c) in a defined stable state due to the assumed all-or-nothing semantics of the failed action,

(d-f) in full operation due to the failure masking effect of the implemented fault tolerance degree.

In cases d-f only local reactions are necessary in order to recover from the failure. The cases a-c require generally global forward recovery procedures to be carried out in addition to a local recovery step for the corresponding object. If the principle of autonomy has been consequently applied to a system design the required global recovery procedure shrinks to a NULL operation.

The above considerations indicate that the proposed object model might might favor extremely simple recovery procedures after a failure. This seems to be another strong argument for the proposed object architecture which needs deeper investigations.

CONCLUSION

The paper proposes an object architecture for distributed realtime systems based on asynchronous message passing. It has been shown that the proposed model

(a) may be used to describe even complex communication and realtime patterns in a simple and elegant way,

(b) is well suited to retain the principle of functional autonomy in the object architecture.

Preliminary considerations on fault tolerance aspects indicate that consequent observance of the principle of functional autonomy will also favor simple recovery procedures.

Forward error recovery is considered the more promising recovery approach for realtime systems. General methods supporting forward error recovery are still missing and need further research investigations.

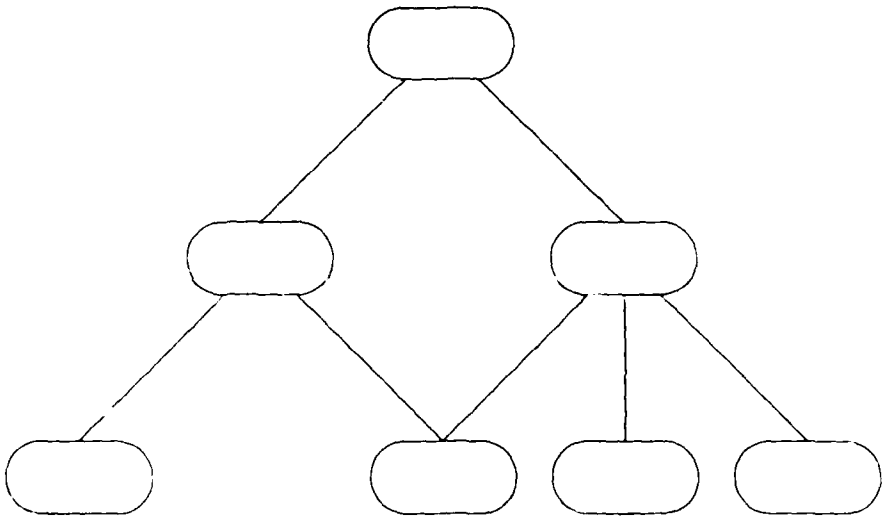
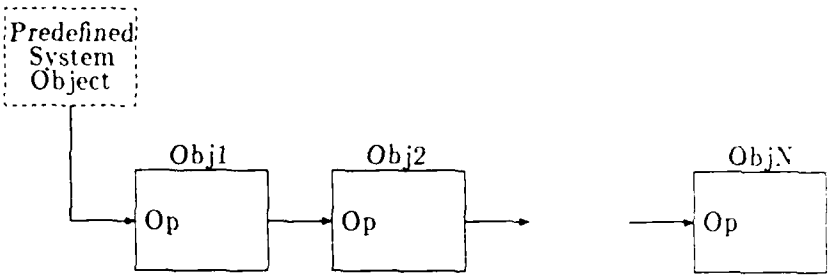
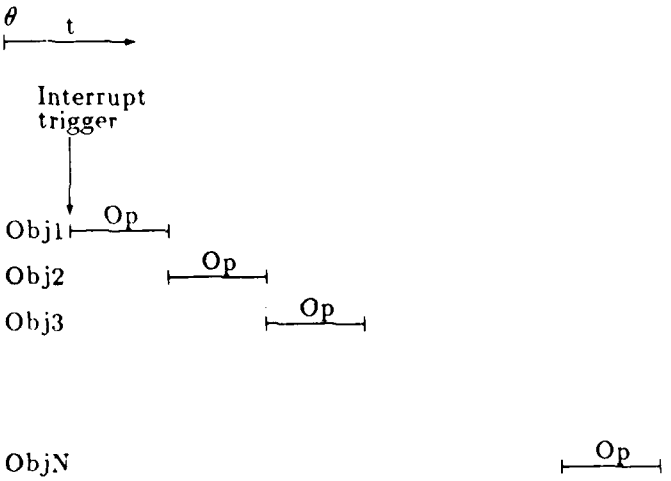


Figure 1 Hierarchical Control Structure of a Process Control System

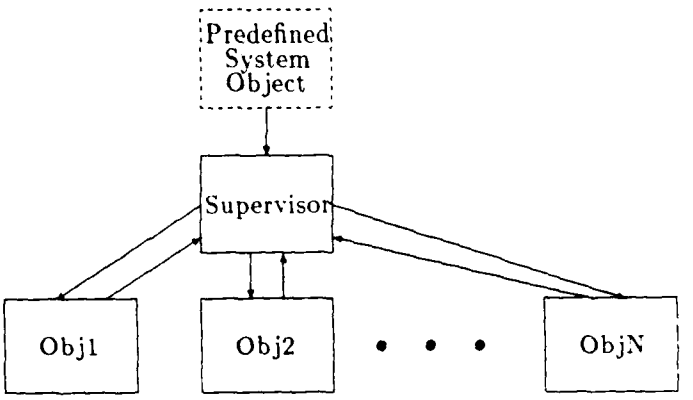


(a) Object interconnection structure

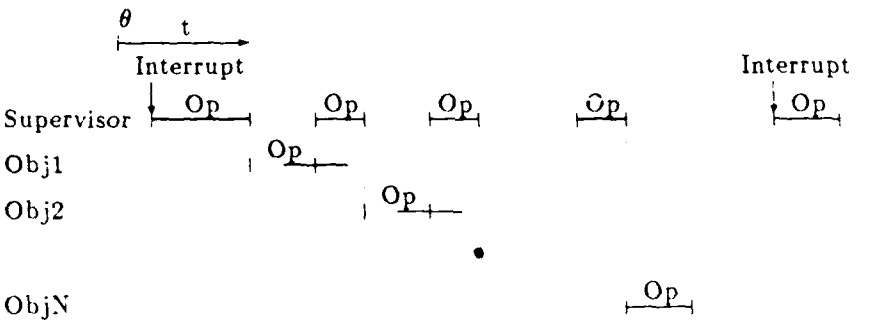


(b) Time diagram

Figure 2 Example of an action sequence with sequencing control included in the Objects Obj1..ObjN

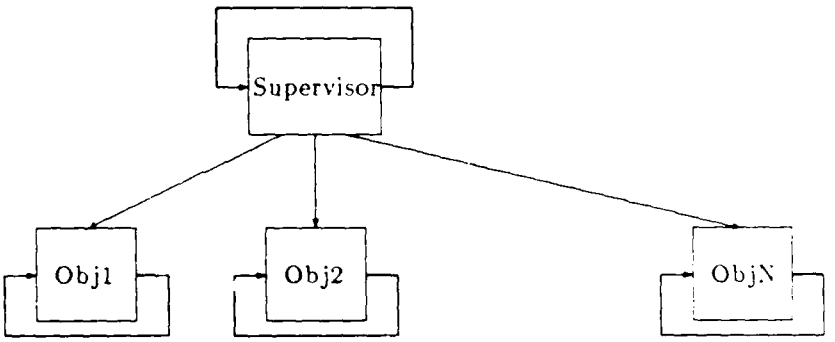


(a) Object interconnection structure

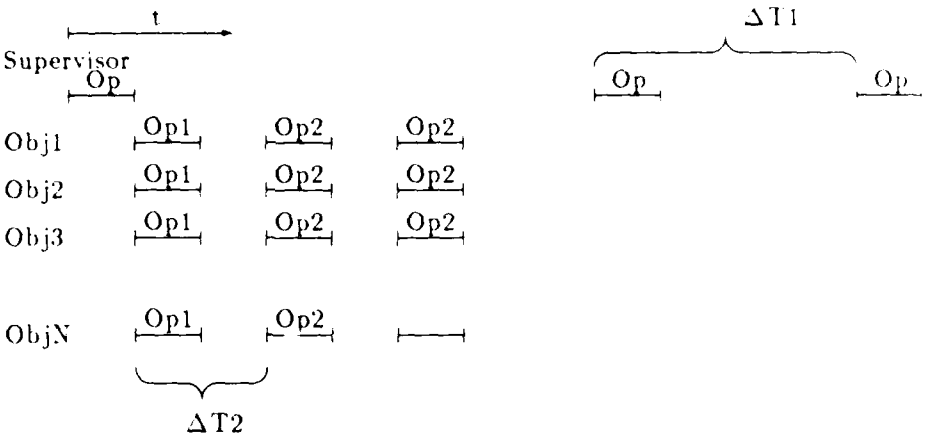


(b) Time diagram

Figure 3 Example of an action sequence with sequencing control included in a supervisory object



(a) Object interconnection structure



(b) Time diagram

Figure 4 A Hierarchical Control Structure with Autonomous Objects Obj1..ObjN

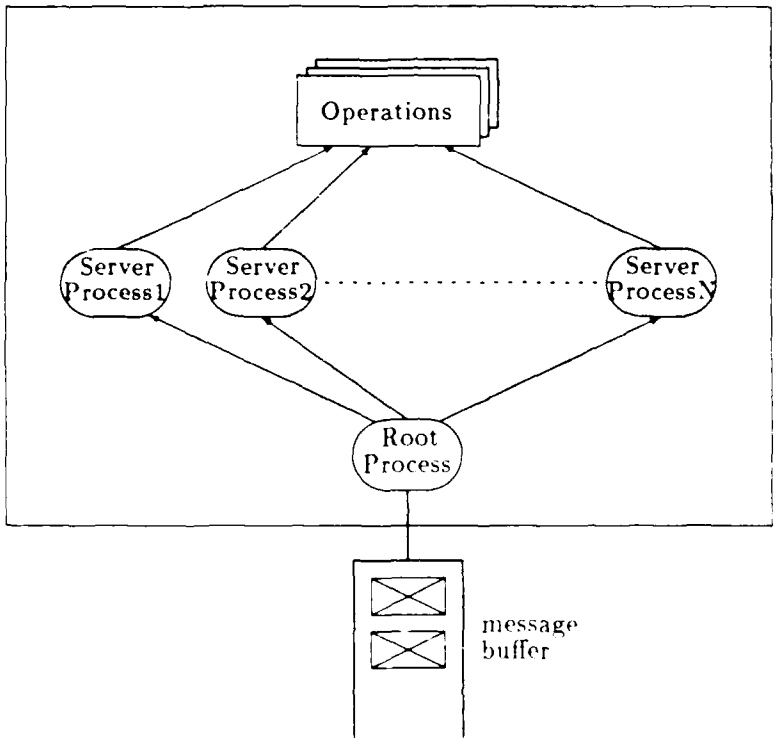


Figure 5 Implementation Structure for Objects Ignoring Specific Reliability Requirements

REFERENCES

[1] A.Agrawala,S.-T.Levi,S.K.Tripathi: The MARUTI Hard Real Time Operating System,Technical Report,Computer Science Dept., Univ. of Maryland 1987

[2] A.Agrawala,S.-T Levi: Temporal Relations and Structures in Real Time Operating Systems,CS-TR-1954,Dept. of Computer Science,Univ. of Maryland,Dec 1987

[3] A.Agrawala,S.-T. Levi: Objects Architecture for Real Time Distributed,Fault Tolerant Operating Systems,IEEE Workshop on Real Time Operating Systems,Cambridge MA,July 1987

[4] G.Agha,C.Hewitt: Concurrent Programming Using Actors-Exploiting Large-Scale Parallelism,in: N.Mahehwari(ED.): 5th Conference on Foundations of Software Technology and Theoretical Comuter Science,p. 19 ff, Springer Verlag LNCS 206(1985)

[5] G.Agha: Actors-A Model of Concurrent Computation in Distributed Systems,MIT AI Lab TR 844(1985)

[6] P.America: POOL-T: A Parallel Object-Oriented Language, in:A Yonezawa,M.Tokoro(Ed.). Object-Oriented Concurrent Programming,MIT Press in Computer Systems(1987)

[7] T.Anderson,P.A.Lee: Fault Tolerance-Principles and Practice, Prentice Hall,London 1981

[8] T.Anderson,J.C.Knight: A Framework for Software Fault Tolerance in Real Time Systems,IEEE-SE 9,No. 3,355-364(1983)

[9] C.Beilken,F.Mattern: The Distributed Programming Language CSSA-A Short Introduction,Report No. 123/85, Dept. of Computer Science,Univ. of Kaiserslautern,FRG (1985)

[10] H-P Boehm,H.L. Fischer,P. Raulefs: CSSA-Language Concepts and Programming Methodology,SIGPLAN Notices Vol. 12,No. 8, 100-108(1977)

[11] D.Cheriton: The V kernel-A Software Base for Distributed Systems,IEEE Software,19-42(1984)

[12] C.Hewitt: Viewing Control Structures as Patterns of Passing Messages,Artficial Intelligence Vol.8,323-364(1977)

[13] R.Koo,S.Toneg: Checkpointing and Rollback Recovery for Distributed Systems,IEEE-SE 13,No.1,23-31(1987)

[14] B.Lampson,H.Sturgis: Atomic Transactions,Lecture Notes in Computer Science 105,246-265,Springer Verlag 1981

[15] B.Liskov,R.Scheifler: Guardians and Actions: Linguistic Support for Robust,Distributed Programs,ACM TOPLAS,381-404(1983)

[16] R.J.Leblanc,C.T.Wilkes: Systems Programming with Objects and Actions,Proc. 5ICDCS,Denver,July 1985

[17] B.Lamson,H.Sturgis: Crash Recovery in a Distributed Storage System,Xerox Palo Alto Research Center,Technical Report April 1979

[18] J.Nehmer et. al.: Key Concepts of the INCAS Multicomputer Project,IEEE-Trans. on Software Engineering,Vol. SE-13,No.8, 913-923(1987)

[19] B.Randell et al.: Reliability Issues in Computing System Design, ACM-Computing Surveys 10,No.2,123-166(1978)

[20] B.Randell: System Structure for Software Fault Tolerance, IEEE-SE 1,No. 2,220-232(1975)

[21] D.L.Russel: State Restauration in Systems of Communicating Processes,IEEE-SE 6,No. 2,183-193(1980)
[22] S.K.Shrivastava,G.N.Dixon,G.D.Parrington: Objects and Actions in Reliable Distributed Systems,Software Engineering Journal Vol.2 No. 5,160-168(1987)
[23] S.K.Shrivastava,L.V.Mancini,B.Randell: On the Duality of Fault Tolerant System Structures,Int. Workshop on "Experiences with Distributed Systems" Univ. of Kaiserslautern, W.-Germany Sept. 1987
[24] L.Svobodova: Resilient Distributed Computing,IEEE-SE 10, No. 3,257-268(1984)
[25] D.J.Taylor: Concurrency and Forward Recovery in Atomic Actions, IEEE-SE 6,No.2,183-193(1980)
[26] A.Ynezawa,M.Tokoro(Ed.): *Object-Oriented Concurrent Programming*, MIT Press in Computer Systems(1987)
[27] Introduction to the iRMX86 Operating System(INTEL Order No. 9803124)
[28] CDOS(Concurrent DOS) System Guide,Doc. No. 1044-2013-002, Version 1.0,Digital Research 1985

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A194 967

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT approved for public release distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION U.S. Army Strategic Defense Command
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science University of Maryland College Park, MD 20742	7b. ADDRESS (City, State, and ZIP Code) Contr & Acq Mgt Ofc, DASD-H-CRS P.O. Box 1500 Huntsville, AL 35807-3801
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DASG60-87-C-0066
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
An Object Architecture for Hard Real Time Systems

12. PERSONAL AUTHOR(S)
Juergen Nehmer

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	March 1988	20

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The paper deals primarily with the question to what extent the object model proposed as a powerful structuring method for large software systems could serve as a suitable basis for the construction of hard distributed real time systems. A particular object model is proposed which supports the idea of object autonomy in a perfect way. It is shown that object autonomy is a key factor to meet extreme real time and fault tolerance requirments.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. Juergen Nehmer	22b. TELEPHONE (Include Area Code) (301) 454-4968 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified