NISTIR 89-4215

NOTE ON NASREM IMPLEMENTATION

John Fiala

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Robot Systems Division
Intelligent Controls Group
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

Lee Mercer, Deputy Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 89-4215

NOTE ON NASREM IMPLEMENTATION

John Fiala

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Robot Systems Division
Intelligent Controls Group
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

December 1989

Issued March 1990

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

Lee Mercer, Deputy Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

Note on NASREM Implementation

Intelligent Controls Group
Robot Systems Division

Principle Author: John Fiala
Date: 12/11/89

Scope of the Document

This document describes ideas formulated in the Intelligent Controls Group for the structure of a NASREM implementation. A basic description of a telerobot control system architecture is given in terms of the model's parallel processing formulation and its meaning in a multiprocessing computing environment. Examples of software organization using Ada are provided to illustrate the ideas.

Keywords: Ada, multiprocessing, control system architecture, software design, real-time control

1. Introduction

The NASA/NBS Standard Reference Model Telerobot Control System Architecture (NASREM) is outlined in [1]. This note discusses how the basic architecture can be decomposed in more detail to arrive at a specific implementation. The discussion emphasizes that the decomposition should maintain certain architectural features with respect to parallel processing and process communication such that the final design accurately represents system parallelism. This type of design provides advantages in building an evolvable system capable of meeting real-time performance requirements.

Figure 1 depicts the basic structure of a telerobot control system architecture. This figure is only meant to show the architecture at a conceptual level. The actual system has much additional structure not revealed at this level. As shown in the figure, the architecture is composed of levels. These levels are given names. The highest level is called the Service Mission level. The lowest level is called the Servo level.

At each "control" level there are sensory processing, world modeling and task decomposition components. Task decomposition components of the architecture are the behavior generators. These elements determine what is to be done and send commands to other system elements to affect that activity. Sensory processing components obtain and process data from system sensors. World modeling components utilize sensory processing data to maintain an internal model of the world. World modeling is the part of the system which interfaces sensory processing and task decomposition activities. Much of what world modeling does has to do with updating the global data system used by sensory processing and task decomposition.

As mentioned above, Figure 1 is conceptual. The figure depicts a system composed of 18 boxes. The figure does not show the level of detail needed for an actual system. For example, the figure does not show how the architecture decomposes around equipment. Figure 1 seems to indicate that the world modeling functions for camera and arm movements are more closely coupled than are the world modeling and task decomposition functions within each of these pieces of equipment. The figure shows <u>one</u> world modeling box and <u>one</u> task decomposition box at each level without showing any separation according to equipment. Clearly, world modeling functions related to arm movements interact much more with arm task decomposition than with eye movement world modeling. Thus, the level of detail of Figure 1 is not adequate for representing the actual system architecture. The appropriate level of detail for a design based on the concepts of [1] is discussed in Sections 2 and 3. Section 4 discusses the attributes of the boxes represented by the system architecture, i.e. how they function as part of a multiprocessing system. Ada implementation examples are given to illustrate the ideas of this section.

2. Decomposition around Equipment

One important feature of the architecture not depicted in Figure 1 is that the architecture hierarchically decomposes around equipment [1, p. 23]. There is a single Task level for an entire robot. This Task level coordinates Elemental-move (E-move) levels for each major equipment subsystem. The E-move levels each coordinate a set of Primitive levels, which in turn coordinate Servo levels for each separate piece of equipment that must be controlled. The levels form a hierarchy (or tree) structure with a single Task level root node and Servo

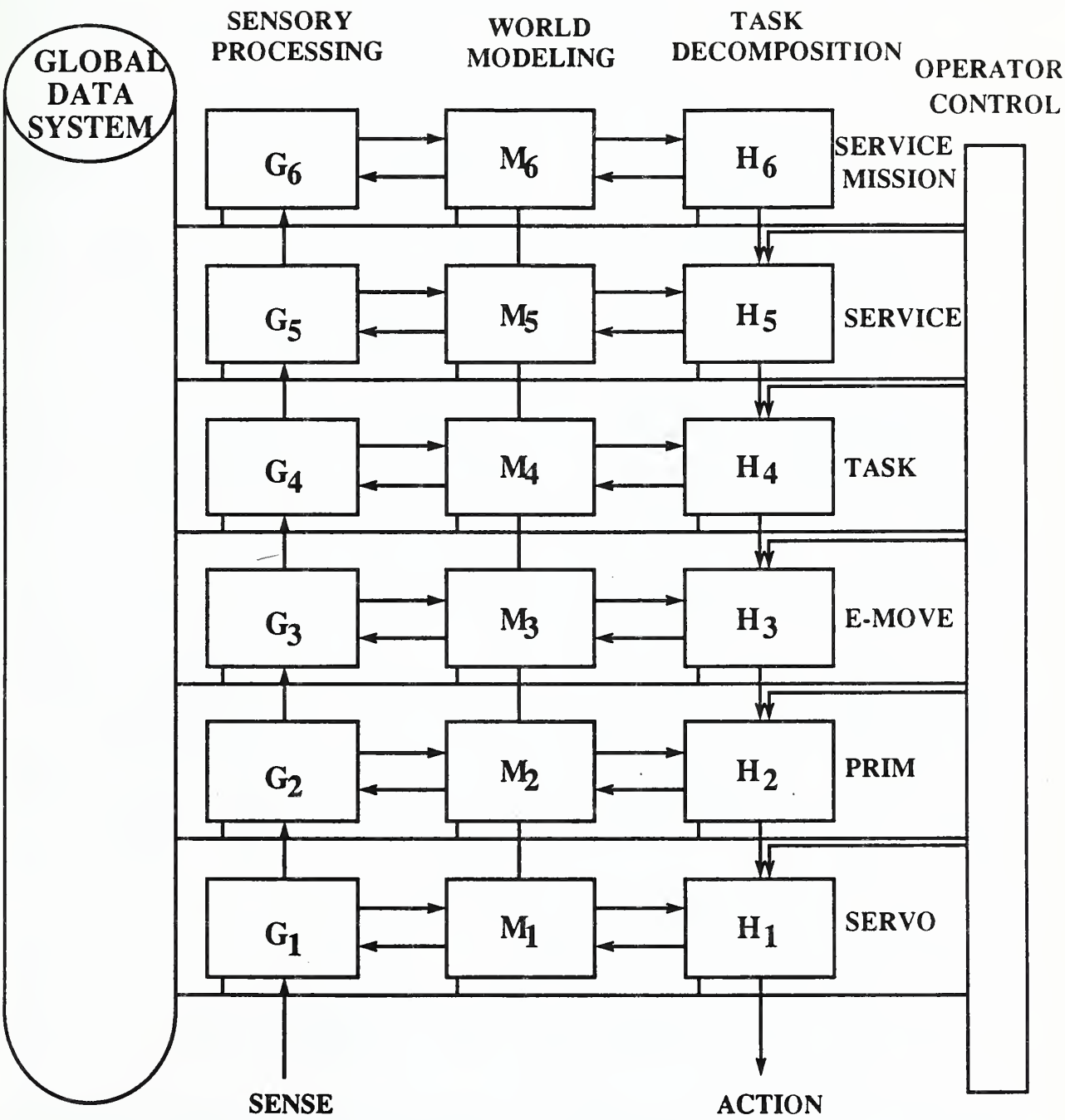

Figure 1. Conceptual Telerobot Control System Architecture.

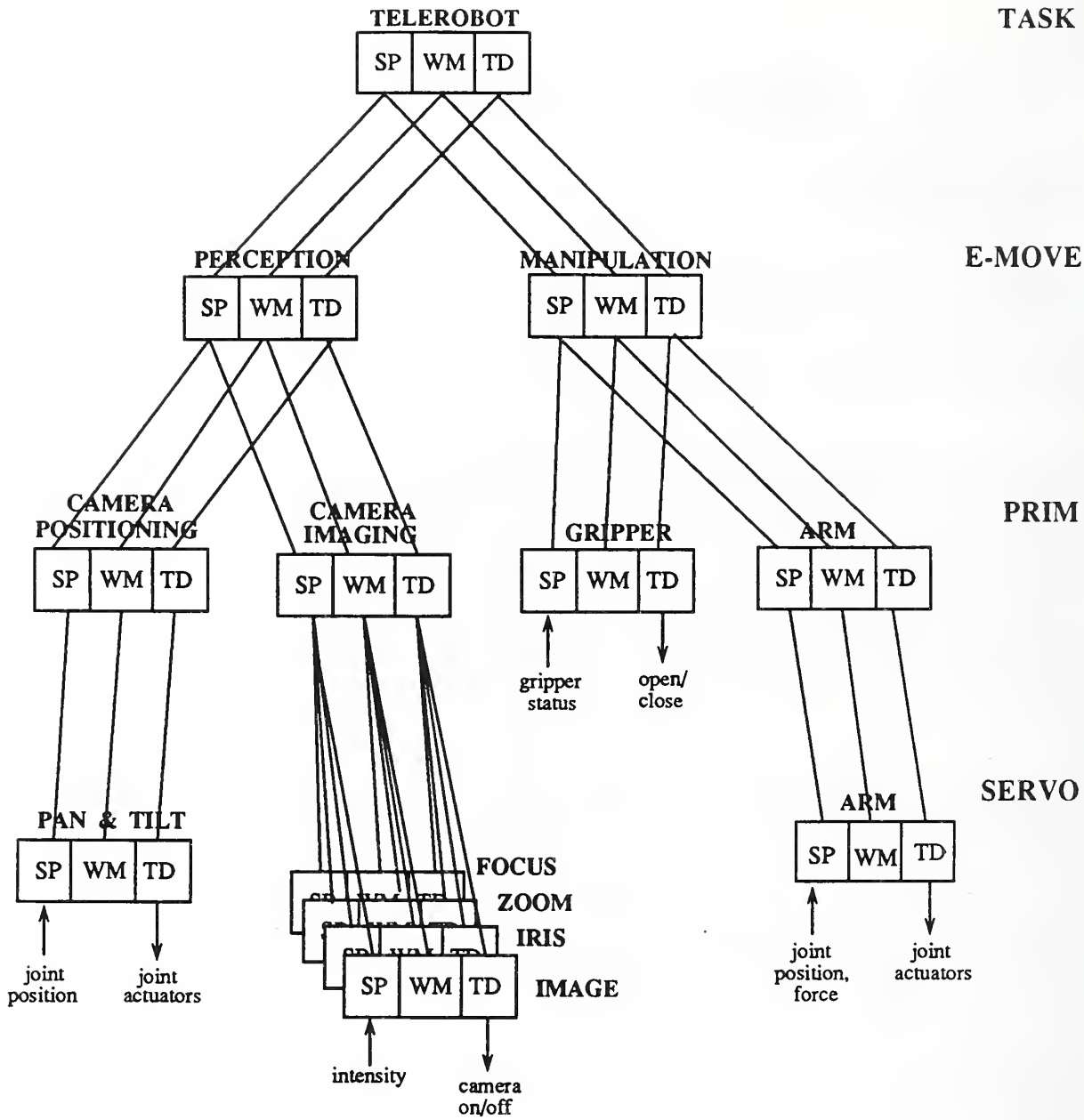

Figure 2. Hierarchical Decomposition around Equipment.

level leaf nodes. As an example of how the architecture decomposes around equipment consider the diagram shown in Figure 2. This diagram depicts a hypothetical robot system that consists of an arm with a simple gripper, and a camera with pan, tilt, zoom, focus and iris control.

There is a Servo level for controlling the manipulator arm. This Servo level contains sensory processing, world modeling and task decomposition components. The Servo level for the arm is commanded by a Primitive level for the arm. (For a more detailed description of the

structure and functionality of manipulator arm Primitive and Servo levels see [2-4].) There is also a Primitive level for the gripper. These Primitive levels have sensory processing, world modeling, and task decomposition components. The gripper and arm Primitive levels are coordinated by an E-move level. If there were more than one arm, the second arm and its gripper would also have Primitive levels under this one E-move level.

There is no Servo level beneath the gripper Primitive level, and the Primitive level communicates directly with the sensors and actuators of the gripper. The gripper in this example is assumed to be controlled open-loop. That is, discrete actions result in the gripper being opened or closed, such as with a pneumatic device. There is no closed-loop control for this device that provides a servo to a commanded position or gripping force. In cases such as this, where there is no closed-loop servo control, it is appropriate to command the device directly from the Primitive level. For the camera device in Figure 2, there are servo levels for the moveable components (pan & tilt, zoom, etc.) because these devices have closed-loop control in the example. If the iris mechanism, for example, were capable only of receiving open-loop commands of aperture settings, then this device would not have a Servo level.

The architecture is determined by the types of control system interfaces provided by the selected equipment. For example, if a simple gripper allowing only discrete open/close commands is used, then the architecture would not have a Servo level for gripper control. However, if a more complicated gripper accepts finger position commands and provides finger position feedback, then an implementation architecture should include a gripper Servo level to servo control gripper opening. The architecure design is dependent on the type of interface provided by the equipment, since this interface completely determines the ability of the control system to influence the behavior of the equipment.

As a further example, consider a gripper that has internal electronics such that it servos to a commanded position using internal position feedback. This feedback is not provided externally, only used internally to close a position loop. Since the interface to this equipment consists of command positions only, it can be considered an open-loop device for the purpose of architecture design. Again, the control system cannot specify the behavior of the local electronics; the behavior is a fixed attribute of the equipment which lies outside the control system architecture.

A requirement of the telerobot architecture is that it be able to evolve and incorporate new capabilities [6]. This implies that individual components be easily modifiable, capable of executing new algorithms and communicating with new system elements. This is not generally a feature of hardware electronics of the type mentioned for the gripper servo above. Since the control system must exhibit many different capabilities, components must also be capable of performing many different algorithms, during normal operation. Again, this eliminates most hardwired control electronics from being included in the system architecture. Although one might label some collection of circuits as being a Servo level for, say, a gripper, this does not give those circuits the properties of multiple capabilities and evolvability that is required of a telerobot control system architecture. Thus, the architecture includes only the programmable components of the control system, the "software" components.

Note that the camera structure includes a Servo level for the image itself. This structure appears to be in conflict with the above mentioned criterion for the existence of a Servo level. Indeed, if the only task decomposition output were "on/off camera" this would be a conflict,

however task decomposition is also involved in planning for and selecting the sensory processing algorithms. Remember that task decomposition determines what is to be done by the system. This means that task decomposition configures the activities within a level, as well as commanding lower levels. Thus, there is a Servo level for configuring and coordinating low-level image processing activities. Due to the complexity of these activities, a level is required to "servo" thresholds and filter parameters for this image processing. (See [5] for details.)

As a final comment on equipment hierarchies it should be mentioned that they can be dynamic. That is, the hierarchy can be reconfigured while the system is in operation [1, p. 6]. As a simple example consider the case where, during normal operation, the manipulator's end-effector is replaced with a new one. The new end-effector may have completely different control requirements such that a new control architecture for this subsystem is mandated. This problem can be accommodated by dynamic reconfiguration of the control hierarchy.

Figure 3 depicts the hierarchical configuration prior to the end-effector replacement. Here, the original gripper "GRIPPER 1" is attached to the arm and is being controlled by the level with a solid box. The structure with dashed-border boxes is inactive, meaning it is not processing. This is the control structure for the replacement gripper "GRIPPER 2" which is not attached to the manipulator. After the end-effector exchange, "GRIPPER 2" is attached to the manipulator and "GRIPPER 1" is not. This situation is depicted in Figure 4, where the "GRIPPER 1" control elements are inactive and "GRIPPER 2" elements are now active. For this example there is probably some amount of time when neither control structure is active, i.e. there is no end-effector attached to the mainpulator, but there is no instant when both are

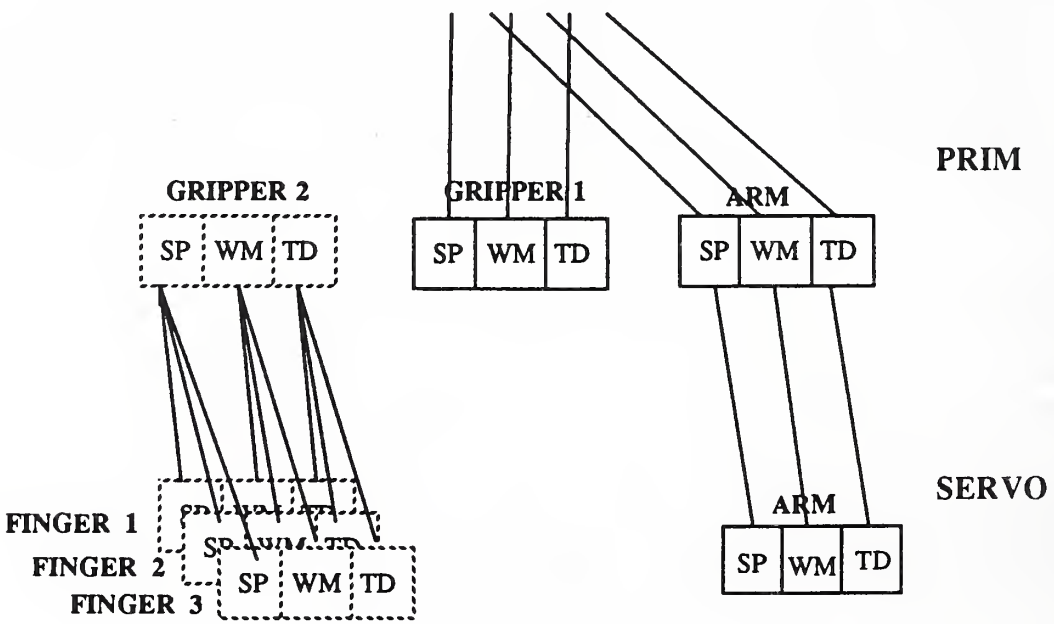

Figure 3. Control Hierarchy with GRIPPER 1 Active, GRIPPER 2 Inactive.

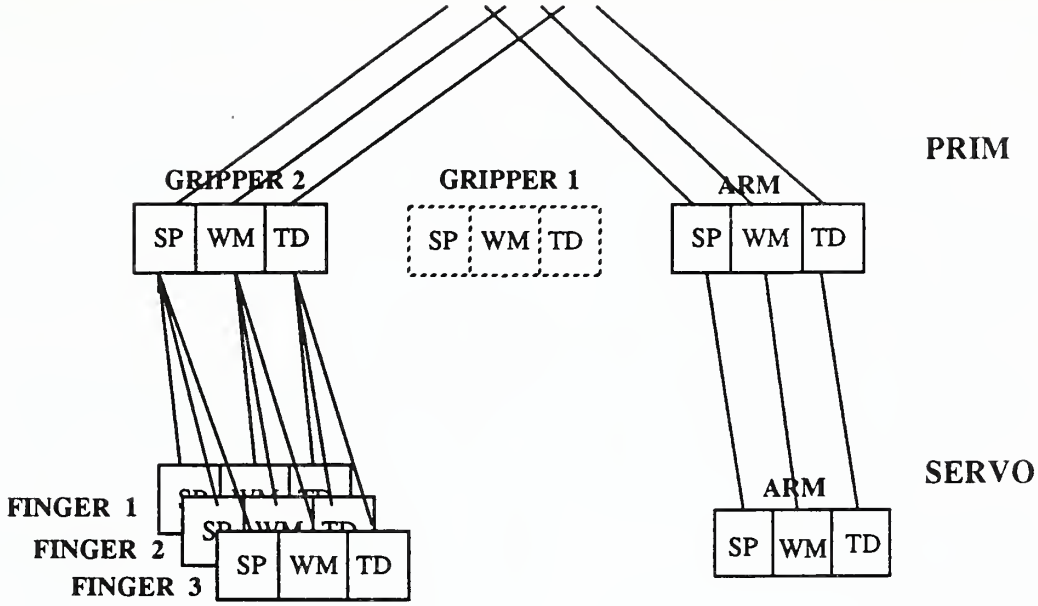

Figure 4. Control Hierarchy with GRIPPER 2 Active, GRIPPER 1 Inactive.

active. The meaning of "active" and "inactive" control elements will be discussed further in Section 4.

3. The Detailed Architecture

A diagram in the form of Figure 2 begins to describe the structure of the control system architecture as designed for a specific implementation. A Figure 2 diagram is implementation-specific because it is determined by the equipment selected for an implementation. Selecting a different set of equipment should result in a different Figure 2 diagram. This section discusses the level of detail required in a final implementation-specific architecture. The level of detail in Figure 2 is not sufficient because the interaction among componenents within a level, and even components between levels, is not specified. An entire level is much too complicated to represent an "atomic" component of the implementation. The following discussion will delineate the characteristics of architecture components such that "atomic" units can be determined. The ultimate goal is to decompose the control system architecture down to this "atomic" level, such that no box can be further decomposed. This level of detail provides a great deal of flexibilty when trying to meet real-time performance requirements, as discussed below.

Given that a decomposition around equipment has been appropriately made in terms of "levels", the next step is to determine the structure within each level. There are task decomposition, world modeling, and sensory processing components within a level, but into how many "boxes" do these general categories of activities decompose? Clearly, this is deter-

mined by the functionality of the specific level, which to a large extent is determined by the decomposition around equipment. This means that decomposition within a level is partly an implementation-specific feature of the architecture. (See Figure 5.)

Reference [1] describes a basic structure for the task decomposition part of a level. Task decomposition breaks into job assignment, planning, and execution modules. There is only one job assignment module for a level, but there may be several planners and executors within a level [1, p. 28]. Each of the task decomposition modules has a well-defined role in the control system as described in [1-3,5]. The job assignment, planning, and execution modules are "atomic" elements of an architecture design by definition.

The criteria of world modeling and sensory processing decompositions is less clearly specified in [1], although some discussion of such decompositions is made in [4]. A principal criterion for these decompositions of sensory processing and world modeling into "atomic" boxes should be "parallelizability" of function, as will be explained subsequently.

The boxes in Figure 1 conceptually represent 18 activities that proceed concurrently. That is, world modeling activities are going on simultaneously with sensory processing and task decomposition activities. The activities at each level also proceed concurrently, with periodic communication between levels serving to coordinate the whole system. For instance, when E-move outputs a new command to Primitive, it does not halt its processing and wait until Primitive completes the command. E-move continues processing while Primitive is acting on the new command. During this time E-move accepts new directives from Task and prepares new outputs for Primitive.

This parallelism of activity holds for the equipment decomposition, as well. For the example of Figure 2, the camera would obtaining new images and adjusting focus and iris concurrently with the movements of the manipulator arm. Within task decomposition, job assignment, planning, and execution activities proceed simultaneously. While the execution box is executing the current plan, the planning box is preparing the next plan, effectively extending the current plan further into the future. Commands entering the task decomposition structure are pipelined from job assignment to planning to execution, such that job assignment, planning, and execution activities are all happening concurrently. Thus, the architecture design is basically a specification of concurrent activities of the control system. This is an important feature of the architecture since it helps to meet the real-time goals as discussed in the next section.

That the final architecture design is to represent the inherent parallelism of the control system activities is the "parallelizability" criterion for motivating decompositions. The sensory processing and world modeling activity at a level should be broken up into boxes such that the function in each box can proceed in parallel with the functions in other boxes. Likewise, the multiplicity of planners and executors within a level must also be determined on the basis of the "parallelizability" of function.

As an example of "atomic" decomposition consider Figure 5. This figure depicts the architectural boxes for task decomposition and world modeling functions for the Servo level of a manipulator arm [2,4]. Each rectangular box is a separate function that can proceed concurrently with the other boxes in the figure. The ovals in the figure represent data that is the output and input to the boxes. (These ovals are components of the "global data system"

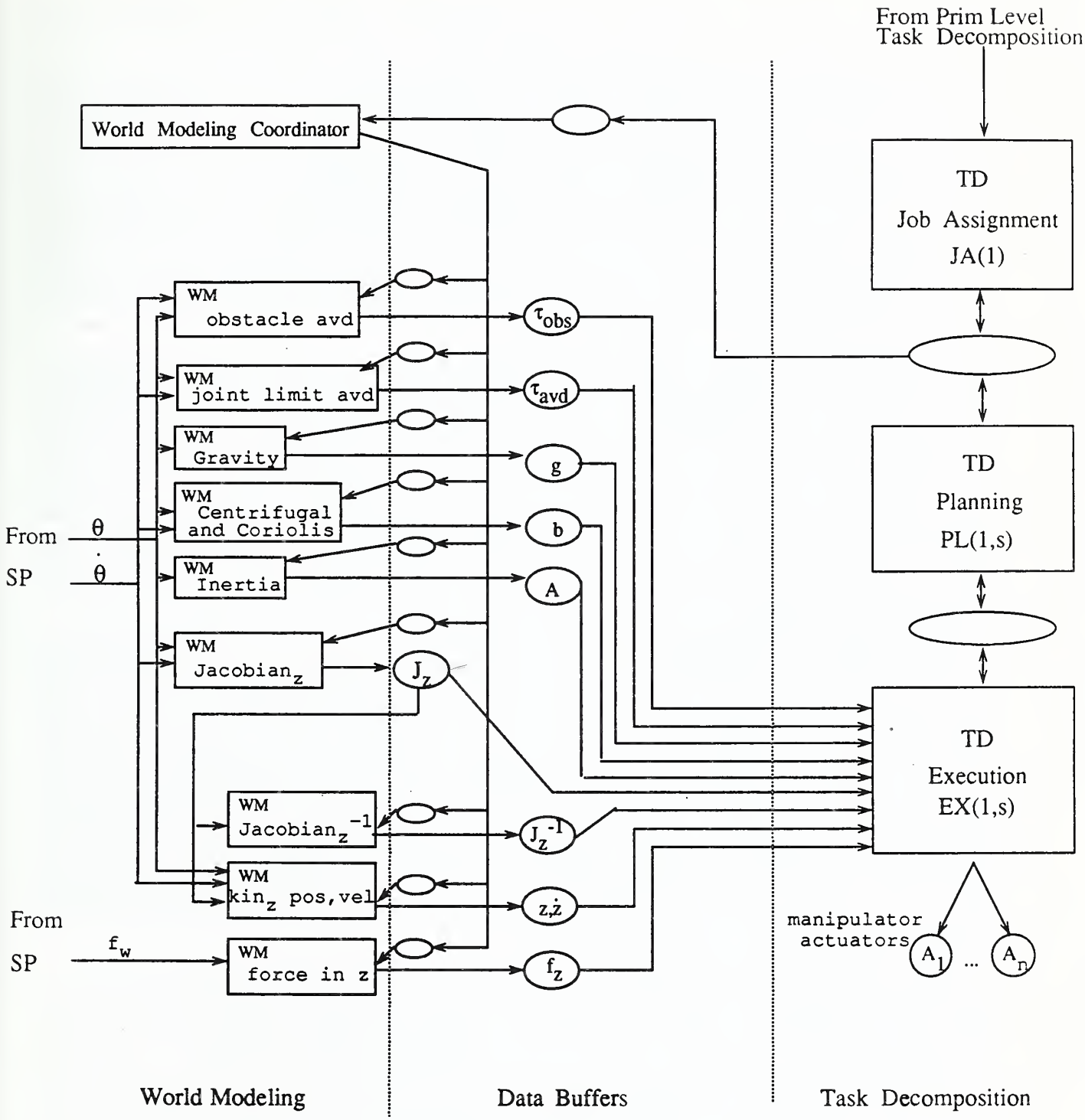

Figure 5. Servo Level Boxes Representing Parallel Functions.

[1,2].) For the example, the boxes shown in the figure are "atomic" units of the Servo architecture.

The figure shows world modeling as composed of several activities which support the task decomposition modules. These activities are concurrent with the task decomposition activities, and, to a large extent, can proceed "asynchronously" with them. This means, for example, that the Jacobian does not have to be computed at a rate directly linked to the rate of the execution module. Jacobians can be computed at some slower rate, with the rate only affecting performance significantly when it is outside some broad range. The time between Jacobian outputs does not even have to be fixed, but can vary from output to output without severely affecting performance. This allowance for "asynchronicity" is another useful criterion for breaking out boxes in the architecture, since this information can also be used to enhance real-time performance as will soon be discussed.

If a function is a separate box, then it will have to interact with the other components of the system by communicating through the global data system. This is a fundamental feature of the control system architecture [1, p. 5]. Unfortunately, global data system communication must come at some expense since it is a special purpose message passing system and not just a transferal of data by, say, a procedural invocation. It is more efficient for a function to operate on local data than to read and write data to the global data system. Thus, one should consider the communication overhead when separating functions into parallel modules. By carefully choosing the decomposition, the communication overhead can be balanced with the speed-up obtained through parallelism.

Suppose an attempt is made to split the Jacobian box in Figure 5 such that two new boxes are generated, one to compute the upper half of the Jacobian and one to compute the lower half. First, both of the new boxes would have to read all the joint angles (in general) such that there is already some cost for this split. It is likely that the elements of the Jacobian share a number of common terms such as sines and cosines of joint angles, and algebraic combinations of these sines and cosines. If each of the new boxes must repeat these calculations it is doubtful that there can be any advantages to the parallelization. If each box computes half of the common terms and shares the results with the other, then a communication bottleneck is reached since this communication is through the global data system. One must conclude that the Jacobian function cannot be effectively decomposed into separate parts and that it should remain as a single box. Through this type of analysis, the "atomic" units of a level can be determined.

This is not to say that a single box can only be executed on a single processor. It is certainly possible, although difficult, to distribute a box over some multiprocessor structure. This would involve developing a communication scheme for the multiprocessor separate from the global data system, since, if the box communicated through the global data system, it would be several architectural boxes instead of one. Developing such specialized communication schemes is difficult and weakens the design since there is less freedom in the distribution of boxes to processors. However, multiprocessor structures are often used in specialized computing hardware, where the communication scheme is already built in.

To conclude, a final control system architecture design is achieved when the design consists only of "atomic" boxes. The decomposition of a level into these "atomic" boxes is based on predetermined functional categories: task decomposition job assignment, task decomposi-

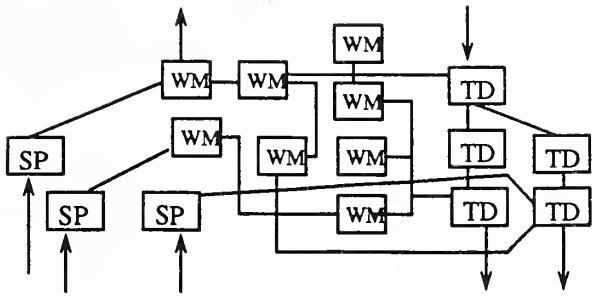

Figure 6. "Atomic" Decomposition for a Level.

tion planning, task decomposition execution, world modeling, and sensory processing. Four other criteria on which to base the separation of activities into separate boxes are:

- Parallelizability
- Asynchronicity
- Communication overhead
- Function decomposeability

Eventually, the architecture design for every level should reach the detail shown in Figure 6, where all interfaces between "atomic" boxes and between boxes and equipment are clearly defined.

4. Properties of "Atomic" Boxes

This section defines the properties of the boxes in the final control system architecture. The description of the architecture [1] is specific with respect to these properties, and any architecture design must adhere to these properties. It is not acceptable to "map" an architecture design onto a completely arbitrary software structure. The "atomic" boxes that have been carefully determined in Sections 2 and 3 represent very specific software entities within the final implementation.

The software entity which implements a box is called a *process* in the following. A process has five important properties:

- Continuous cyclic execution
- Read-compute-write execution cycle
- Concurrency
- Interfaces through global data system
- Inactivation

First, a process repeatedly performs its designated function in the system. Secondly, each *execution cycle* of a process consists of the sequence, read inputs-perform computation-write outputs. A process that repeatedly performs this cycle is said to be *cyclicly executing*.

Since a process can run concurrently and (sometimes) asynchronously with the other processes of the system, a process should be capable of retaining some context from cycle-to-cycle in the form of *process variables*. Process variables are not globally defined. The fourth property of a process, however, is that the inputs and outputs are via the global data system. Finally, a process can be made *inactive*, as described below.

For most processes of the system, the read-compute-write cycle is continually performed for the life of the system. Even when no new commands are issued to a process, cyclic execution is still being performed. This allows the robot to react to sensed changes in the environment, e.g., disturbances to the servo control loop. In order to insure that a process will be able to react quickly, the process should not deadlock waiting for communication with any one process. Thus, reliable cyclic execution requires that a process be able to read or write data with as little waiting as possible. A process should always read and execute on the most current data that is available.

The data that a process reads (writes) is obtained from (sent to) the global data system. It is through the global data system (and only through the global data system) that a process communicates with other processes. Thus, the lines depicting interfaces between boxes in Figure 6 represent global data system communication links. The global data system must protect the data from corruption while allowing global access to the data. Typically an interface written by one process is read by several other processes so that the global data system must accommodate simultaneous accesses to the same data. There are a number of techniques for implementing this type of data system, however they are not discussed in this document.

As described in Section 2, a process can be made inactive. This means that the process no longer executes read-compute-write cycles. The process is removed from the set of cyclicly executing processes (active processes) that compose the control system. Processes that share the use of one processor obviously cannot run simultaneously. However, a process which gives up the processor to allow another to execute in some time-sharing scheme is not "inactive" as defined here. Multiprocessing schemes which allow several processes to share a processor fairly should not disturb the apparent concurrency of activity as seen by the rest of the system. For example, a simple scheme would let each process execute one cycle in sequence, repeating the sequence of one-cycle process execution indefinitely. An inactive process would be one that is removed from the sequence for a period of time, such that it no longer is cycling with the others.

Since every box in the architecture design represents a software process that can run in parallel with the other processes (boxes) of the system, the allocation of a process to a processor should be essentially arbitrary. This is provided of course that the processor's computing power is sufficient and that the box has been well-enough designed to have reasonable communication overhead. With this in mind, it is desireable to develop a software model of a process that would allow it to be easily moved from one processor to another. There are many ways to do this. Some languages, such as Ada, provide a parallel process model inherent to the language. In Ada, this is the "Tasking" model. Even in Ada, however, there are other possibilities for process models which would achieve the desired objective.

As an example, consider the following model. A process is defined that consists of three elements, an initialization procedure, a process body, and a set of process variables. The ini-

tialization procedure for a process runs before the process body runs. It initializes the process variables and their common memory areas, and performs any other initialization required by the process. The process body consists of a read-compute-write cycle which can be executed repeatedly to carry out the functions of the process. Each execution cycle, the body reads its input buffers, computes outputs, and writes the output buffers. The process variables are the data buffers which are read and written by the process body.

Incorporating this model in an Ada package, the package specification is

```
package YURBOX_PROCESS is

    procedure YURBOX_INIT;

    procedure YURBOX;

end YURBOX_PROCESS;
```

where YURBOX is the name of the process being implemented. YURBOX_INIT is the initialization procedure and YURBOX is the process body. The process variables are hidden within the package body so that they are only visible to the process itself. Thus, the variables are local to the process but retain their values between process body cycles. It is assumed that the global data system declarations of the interfaces is made elsewhere

```
with PROCESS_COMMAND, PROCESS_STATUS;
use  PROCESS_COMMAND, PROCESS_STATUS;
package body YURBOX_PROCESS is

    YURBOX_CMD: YURBOX_CMD_TYPE;
    YURBOX_STAT: YURBOX_STATUS_TYPE;

    procedure YURBOX_INIT is separate;

    procedure YURBOX is separate;

end YURBOX_PROCESS;
```

Here, PROCESS_COMMAND and PROCESS_STATUS contain the command and status data structures. The structures are instantiated as process variables YURBOX_CMD and YURBOX_STAT. Thus, the process body will read commands from global memory into YURBOX_CMD, and write status into global memory from YURBOX_STAT.

The process body can also be implemented as an Ada task. This may be desirable because of features provided by tasks which are not available to procedures. To implement as a task, the procedure declaration in the specification is replaced with

```
task YURBOX_TASK is
  entry CYCLE;
end YURBOX_TASK;
procedure YURBOX renames YURBOX_TASK.CYCLE;
```

and the declaration in the package body is replaced with

```
task body YURBOX_TASK is separate;
```

This type of task declaration allows a main procedure to invoke either the procedure version or the task version of the process body via the statement YURBOX;. However, a small change is required within the task body itself. The task body must contain a loop around the whole read-compute-write sequence so that the task never terminates. By putting an accept statement at the top of the loop, a single loop iteration can be invoked by the main, just as for the procedure definition. Thus, the task body would look like

```
task body YURBOX_TASK is
 :
begin

  loop
    accept CYCLE;

    <do reads>
    <call procedure to compute outputs>
    <do writes>
  end loop;

end YURBOX_TASK;
```

Each processor board in the system would have a main procedure. The main procedure would first call the initialization procedures of the processes that are to run on the board, then call the process bodies. For example,

```
with YURBOX_PROCESS;  use YURBOX_PROCESS;
procedure YURMAIN is
begin

   YURBOX_INIT;

   loop
      YURBOX;
   end loop;

end YURMAIN;
```

In this example, the main initializes YURBOX and then repeatedly invokes the process body. Note that YURBOX could be implemented either as a task or a procedure in this example - the main procedure does not change. However, it is possible to have the process body implemented as a task such that the accept CYCLE; statement precedes the loop. In this case, the task will handle do its own looping via the Ada tasking model. A main for this type of task would look like

```
with YURBOX_PROCESS;  use YURBOX_PROCESS;
procedure YURMAIN is
begin

    YURBOX_INIT;

    YURBOX;

end YURMAIN;
```

All of the above software structures are similar enough that processes can be easily moved from one processor to another. The only code that is modified (once the processes are written) is the board main procedures. Then it is simply a matter of remaking the executable images that will be placed on the processor boards.

A simple way to handle process inactivation can be used with the above process model. This technique involves having each process read a data area that indicates whether it should run or not. This is depicted for world modeling in Figure 5. Here, a World Modeling Coordinator specifies which processes of world modeling are to be active. The coordinator writes into a global data area for each process indicating whether the process is to cycle or remain dormant. Ada code for a procedure process body using this method of inactivation might look like

```
procedure YURBOX is
   :
begin

   READ( ACTIVE_FLAG );
   if not ACTIVE_FLAG then return;
   end if;

   <do reads>
   <call procedure to compute outputs>
   <do writes>

end YURBOX;
```

Here, a procedure READ is assumed that obtains the data item ACTIVE_FLAG from the global data system. If ACTIVE_FLAG is not set, the process is not supposed to perform its

read-compute-write cycle, and so it returns to the main directly.

Note that the model using procedures for process bodies runs each process cycle in a fixed sequence. This is one of the simpliest schemes for concurrency on a single processor. More complicated execution of process cycles can be achieved in the model by using Ada tasks to implement process bodies. There are myriad ways to implement the software processes of the control system. The Ada code provided here is only an example. However, any implementation must adhere to the properties of a process mentioned above, as these properties assure that the architecture truly represents system parallelism.

5. Conclusions

Having an architecture design and process model as described here can greatly enhance our ability to meet the real-time requirements of the control system. The architecture is designed such that it represents the inherent parallelism of the control system functions, minimizing communication overhead while allowing the maximum utility of the components. The implementor can use this design to determine how to distribute processes to processors such that the overall performance requirements are met.

A box in a final telerobot control system design CANNOT be broken up into smaller processes, each of which could be assigned to different machines. It must be "atomic", or the design is not complete. A design can change such that what was once a single process is now implemented by several read-compute-write processes. However, such a change constitutes a new control system design with respect to NASREM, i.e., a different NASREM implementation.

Clearly, a NASREM design reflects decisions made by the designer specific to his or her implementation. There is no one "correct" design, however there are obvious attributes that make one design better than another. A good design should be able to incorporate new capabilities easily and should facilitate the ability to meet real-time performance requirements. In addition, the design should reflect how the actual system is decomposed into parallel processes as defined here.

6. References

[1] Albus, J.S., McCain, H.G., Lumia, R., "NASA/NBS Standard Reference Model Telerobot Control System Architecture (NASREM)," NIST Technical Note 1235, NIST, Gaithersburg, MD, July, 1987.

[2] Fiala, J., "Manipulator Servo Level Task Decomposition," NIST Technical Note 1255, NIST, Gaithersburg, MD, October, 1988.

[3] Wavering, A. "Manipulator Primitive Level Task Decomposition," NIST Technical Note 1256, NIST, Gaithersburg, MD, October, 1988.

[4] Kelmar, L. "Manipulator Servo Level World Modeling," NIST Technical Note 1258, NIST, Gaithersburg, MD, March, 1989.

[5] Chaconas, K., Nashman, N., "Visual Perception Processing in a Hierarchical Control System: Level 1," NIST Technical Note 1260, NIST, Gaithersburg, MD, June, 1989.

[6] Goddard Space Flight Center Internal Document, "Short Term Evolution for the Flight Telerobotic Servicer," GSFC, April, 1989.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NISTIR 89-4215	2. Performing Organ. Report No.	3. Publication Date MARCH 1990

4. TITLE AND SUBTITLE

Note on NASREM Implementation

5. AUTHOR(S)

John C. Fiala

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
U.S. DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

S/A

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

This document describes ideas formulated in the Intelligent Controls Group for the structure of a NASREM implementation. A basic description of a telerobot control system architecture is given in terms of the model's parallel processing formulation and its meaning in a multiprocessing computing environment. Examples of software organization using Ada are provided to illustrate the ideas.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

Ada; control system architecture; real-time control; software design

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

20

15. Price

A02

USCOMM-DC 6043-P80