(NASA-TM-89375) GENERAL OBJECT-ORIENTED N87-23190
SOFTWARE DEVELOPMENT (NASA) 97 p Avail:
NTIS HC A05/MF A01; single copies
available from NASA/GSFC, Code 552, Unclas
Greenbelt, Md. 20771 CSCL 09B G3/61 0076838

SOFTWARE ENGINEERING LABORATORY SERIES SEL-86-002

GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT

AUGUST 1986

NASA

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

The primary contributors to this document are

Ed Seidewitz	(Goddard Space Flight Center)
Mike Stark	(Goddard Space Flight Center)

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

0252

ABSTRACT

Object-oriented design techniques are gaining increasing popularity for use with the Ada programming language. This report describes a general approach to object-oriented design which synthesizes the principles of previous object-oriented methods into a unified framework. Further, this approach fits into the overall software life-cycle, providing transitions from specification to design and from design to code. It therefore provides the basis for a general object-oriented development methodology.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Cont'd)

SECTION 1 - INTRODUCTION

An "object" is an abstract software model of a problem domain entity. Objects are packages of both data and operations on that data [Goldberg 83, Booch 83]. The Ada[1] package construct is representative of this general notion of an object. "Object-oriented design" is the technique of using objects as the basic unit of modularity in system design. The Software Engineering Laboratory at the Goddard Space Flight Center is currently involved in a pilot project to develop a satellite dynamics simulator in Ada (approximately 40,000 statements) using object-oriented methods [Agresti 86, Nelson 86]. Several authors have applied object-oriented concepts to Ada (e.g., [Booch 83, Cherry 85b]). These methods are useful, but limited when considered as a general approach to developing large software systems [Nelson 86]. As a result we have synthesized a more general approach which allows a designer to apply powerful, object-oriented principles to a wide range of applications and at all stages of software development. This report describes our approach and considers how object-oriented design fits into the overall software life-cycle.

The present report supercedes and expands our earlier work on this topic [Seidewitz 85a, Seidewitz 85b, Seidewitz 86, Stark 86]. However, our work is still in progress and future versions of this document will include material on object-oriented specification and testing.

[1]Ada is a trademark of the U.S. Government (Ada Joint Program Office).

SECTION 2 - PROCEDURES AND OBJECTS

In object-oriented design, the basic unit of modularity is the object rather than the procedure. While a procedure defines a specific operation, an object defines a "state machine" with internal memory and multiple operations on that memory. This section discusses the concepts of objects and procedures and shows the relationship between them.

2.1 PROCEDURES

We begin with the more familiar concept of a procedure. We can model a procedure as a mathematical function. Figure 2-1 shows one possible diagram for representing such a function. In this diagram, the arrows represent data flows into and out of the procedure. However, in a computer program, there is a flow of control as well as data. Thus, when a procedure is called, we can say that control "flows into" the procedure. When the procedure is complete, control returns to the caller.

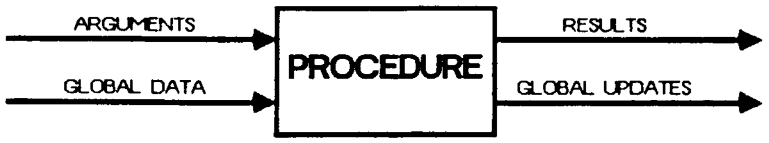


Figure 2-1. Procedure Data Flow

The diagram in Figure 2-2 shows both the data flow and the control flow. The arrow from CALLER into PROCEDURE indicates that CALLER transfers control to PROCEDURE. Note that the return of control to CALLER is not explicitly shown, but is assumed to happen when PROCEDURE is finished. The smaller arrows along the larger control flow arrows show the data flows (similar to [Yourdon 79]), which may go in either direction along a control path. Also notice that in Figure 2-2 we have added an explicit symbol for the GLOBAL DATA. Although control never really flows into data, we show access to such data symbols by arrows always directed toward the data. This indicates that the data is always passive and never initiates any action.

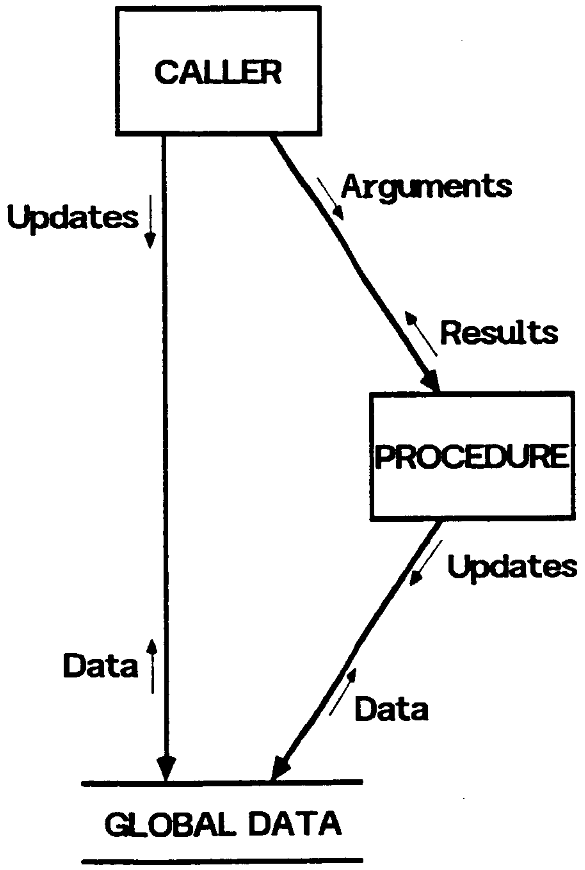


Figure 2-2. A Procedure Call, With Data Flows

When there are several control paths on a diagram, each with several data flows, showing all data flows can become cumbersome. Therefore, instead of explicitly showing the data flow on the diagrams, we include the data flow in a separate "operation definition" which describes the operation provided by the procedure. Figure 2-3 shows the diagram of Figure 2-2 redrawn without the data flow arrows. The operation definition for the procedure in Figure 2-3 would be:

```
    PROCEDURE (ARGUMENTS) RESULTS
```

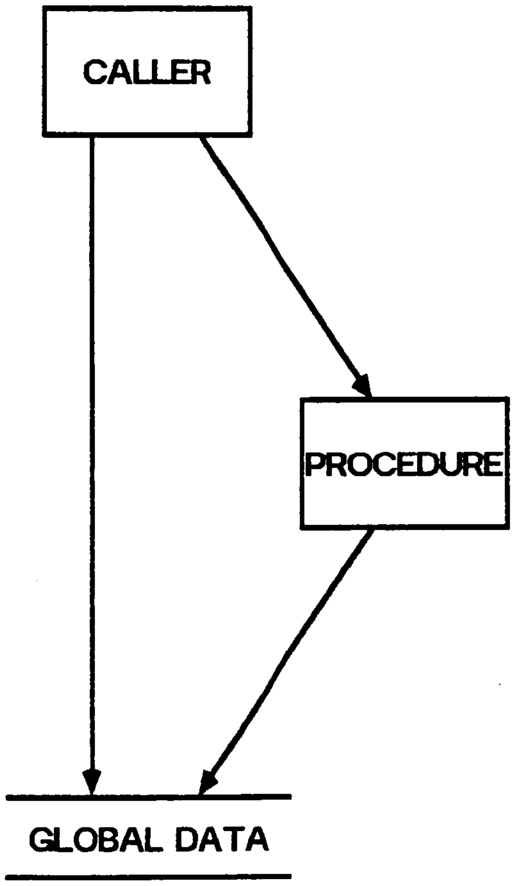


Figure 2-3. A Procedure Call, Without Data Flows

In the operation definition, the parenthesized data flows with the control flow, while the unparenthesized data flows against the control flow. A general operation thus includes two data flows. However, some operations have only one data flow, which may be in either direction relative to the control flow. In fact, some operations simply signal an action with no data flow at all. Such operations have definitions of the form:

```
    RESET ()
```

The parentheses are included even when they are empty. For data symbols such as GLOBAL DATA in Figure 2-3, the control arrows implicitly define the appropriate data flows with no need for operation definitions.

2.2 OBJECTS

Whatever the notation, we still model a procedure as a mathematical function. That is, given a certain set of inputs (arguments and global data), a procedure always produces the same set of outputs (results and global updates). A procedure, for example, cannot directly model an address book, because an address book has memory (a set of addresses) which can be accessed and updated. Normally, the solution to this is to place this memory in global variables, leaving it exposed to illicit modification.

An object, on the other hand, packages some memory along with all allowable operations on it. We can model an object as a mathematical "state machine" with some internal state which can be accessed and modified by a limited number of mathematical functions. We thus implement an object as a packaged set of procedures and internal data, as shown in Figure 2-4. For an address book object, the internal memory would be a set of addresses, and the allowable operations would be accessing an address by name, adding a new address, etc.

Internally, the procedures in an object are functions of both arguments and the internal memory. Externally, however, an object appears as a "black box" with operations on certain arguments producing certain results. Now, though, the same arguments may produce different results at different times, depending on the hidden internal state. An "object description" includes a list of definitions for each of the operations provided by the object. For example:

ADDRESS-BOOK

Provides:

ADD (NAME + ADDRESS)
CHANGE (NAME + ADDRESS)
LOOKUP (NAME) ADDRESS
REMOVE (NAME)

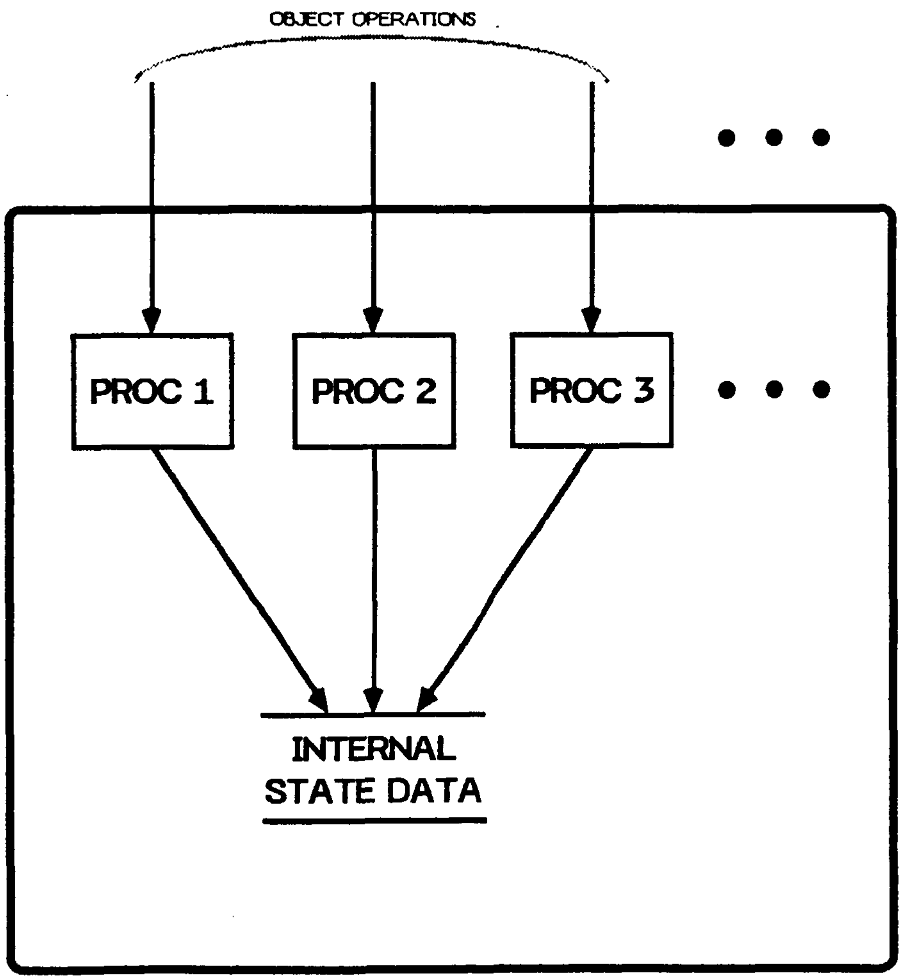


Figure 2-4. An Object

An object can also represent a "type manager." A "type" is basically a template for a set of objects which all allow the same operations. The "type manager" object combines in its own state one complete state for each object of the type. Each type operation is then augmented with a data item which selects the specific object (state) to be operated on. For example, we could define a type manager which would allow creation of an arbitrary number of address books. The new object definition would be:

```
    ADDRESS-BOOK-MANAGER

         Provides:

              ADD (ADDRESS-BOOK + ADDRESS + NAME)
              CHANGE (ADDRESS-BOOK + ADDRESS + NAME)
              LOOKUP (ADDRESS-BOOK + NAME) ADDRESS
              REMOVE (ADDRESS-BOOK + NAME)
              CREATE () ADDRESS-BOOK
```

The new data item ADDRESS-BOOK must be specified for each address book operation, and the new operation CREATE returns a new, empty ADDRESS-BOOK.

SECTION 3 - OBJECT DIAGRAMS

In this section we will connect objects into "object diagrams" which represent system designs. Operations must take place between two objects, with control flowing from one to the other. Such a connection of two objects is called a "communication." In a communication, control flows out of one object to "invoke" an operation. The object which receives the flow of control then "services" this operation. The point of an object diagram is to show all possible communications in a system.

3.1 NOTATION

As an example, consider a simple schedule organizer that consists of three objects: a USER INTERFACE, an ADDRESS BOOK and a DATE BOOK. Figure 3-1 shows a possible object diagram for this system. The round-cornered squares in Figure 3-1 represent objects. The arrows between objects represent communications. Note, however, that each arrow can represent a call on one or more operations provided by the object to which it points. For each arrow leaving an object, we add to that object's description a list of operations used from the other object. For example, the object descriptions for the objects in Figure 3-1 are:

```
USER-INTERFACE

     Provides:

          RUN ()

     Uses:

          TERMINAL
            GET
            PUT
```

ADDRESS-BOOK
- ADD
- CHANGE
- LOOKUP
- REMOVE

DATE-BOOK
- GET-APPOINTMENT
- MAKE-APPOINTMENT
- CANCEL-APPOINTMENT

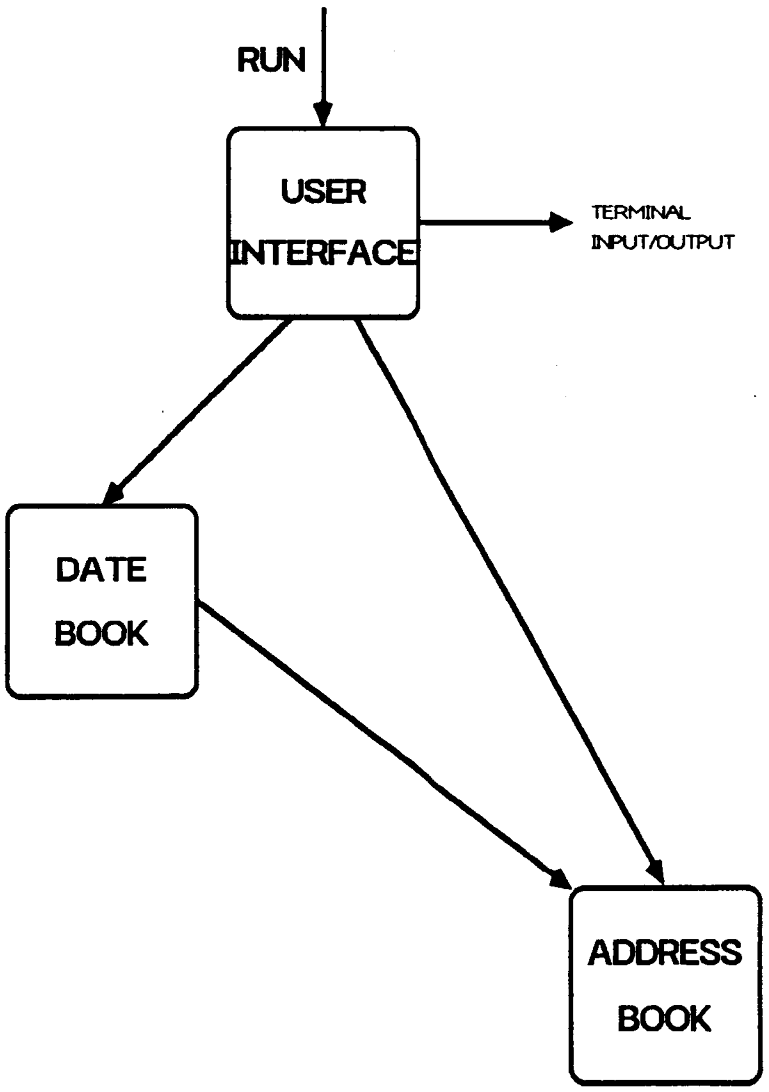


Figure 3-1. Schedule Organizer Object Diagram

```
DATE-BOOK

     Provides:

          GET-APPOINTMENT (DATE + TIME) NAME + ADDRESS
          MAKE-APPOINTMENT (DATE + TIME + NAME)
          CANCEL-APPOINTMENT (DATE + TIME)

     Uses:

          ADDRESS-BOOK
            LOOKUP

ADDRESS-BOOK

     Provides:

          ADD (NAME + ADDRESS)
          CHANGE (NAME + ADDRESS)
          LOOKUP (NAME) ADDRESS
          REMOVE (NAME)
```

The user communicates with this system through the USER INTERFACE object. The system allows the user to store and retrieve addresses in ADDRESS BOOK. The user can also schedule appointments in his DATE BOOK with people he knows. When the user requests to see what appointment is scheduled at a certain time, DATE BOOK also automatically retrieves the address of the person to be met. The object diagram shows all the communications necessary to perform these functions. It thus defines the objects needed in the system and all the interactions between these objects.

In the above example, it is fairly easy to see that the "main control" object is USER INTERFACE. The only operation serviced by USER INTERFACE is the operation RUN. This operation is used to invoke the system, passing control into USER INTERFACE. USER INTERFACE then passes control to the other objects as necessary to perform the functions of the system. Thus all the other control flows are out of USER INTERFACE.

By convention, the arrow representing the initial flow of control into a system is labeled "RUN" on an object diagram, as shown in Figure 3-1.

So far we have been thinking of operations and communications as modeling the traditional procedure call/return mechanism. Communications can, however, represent more then just simple procedure calls. They may also model an Ada entry call and rendezvous. In this case, it may not be so obvious which way control should flow. Consider the example shown in Figure 3-2(a), with the following object descriptions:

```
DATA-ACCUMULATOR

     Provides:

          RUN ()

     Uses:

          SOURCE
            GET

          PACKET-TRANSMITTER
            SEND

PACKET-TRANSMITTER

     Provides:

          RUN ()
          SEND (DATA-PACKET)

     Uses:

     DATA-LINE
       TRANSMIT
```

The control flow of the RUN communication branches and flows into both of the objects in Figure 3-2(a). This means that there is a thread of control in both objects at the same time. That is, they run concurrently. In this example, the DATA ACCUMULATOR gathers real-time data from some ongoing experiment into fixed size packets. These packets are then transmitted along a data line to a remote laboratory by PACKET TRANSMITTER. Figure 3-2(a) shows that when the DATA ACCUMULATOR has accumulated enough data to form a packet, it initiates a communication with PACKET TRANSMITTER to hand over the packet to be transmitted. Note that in the case of concurrent objects, one object may have to wait before an operation it invokes is serviced. Section 3.3 will consider this further. An alternative design is shown in Figure 3-2(b). The new object descriptions are:

```
DATA-ACCUMULATOR

     Provides:

          RUN ()
          GET-PACKET () DATA-PACKET

     Uses:

          SOURCE
            GET

PACKET-TRANSMITTER

     Provides:

          RUN ()

     Uses:

          DATA-ACCUMULATOR
            GET-PACKET

          DATA-LINE
            TRANSMIT
```

Because control resides simultaneously in both objects, either object can initiate communications. In Figure 3-2(b), when the PACKET TRANSMITTER is ready to transmit a new packet, it initiates a communication with the DATA

ACCUMULATOR. When the DATA ACCUMULATOR services this operation, a DATA-PACKET is passed to the PACKET TRANSMITTER. In this example, both designs are equally good. In more complicated concurrent systems, there are various reasons for choosing one direction of control flow over the other. In any case, to change the design in this way requires a change in the direction of an arrow on the object diagram and the modification of the appropriate object definitions.

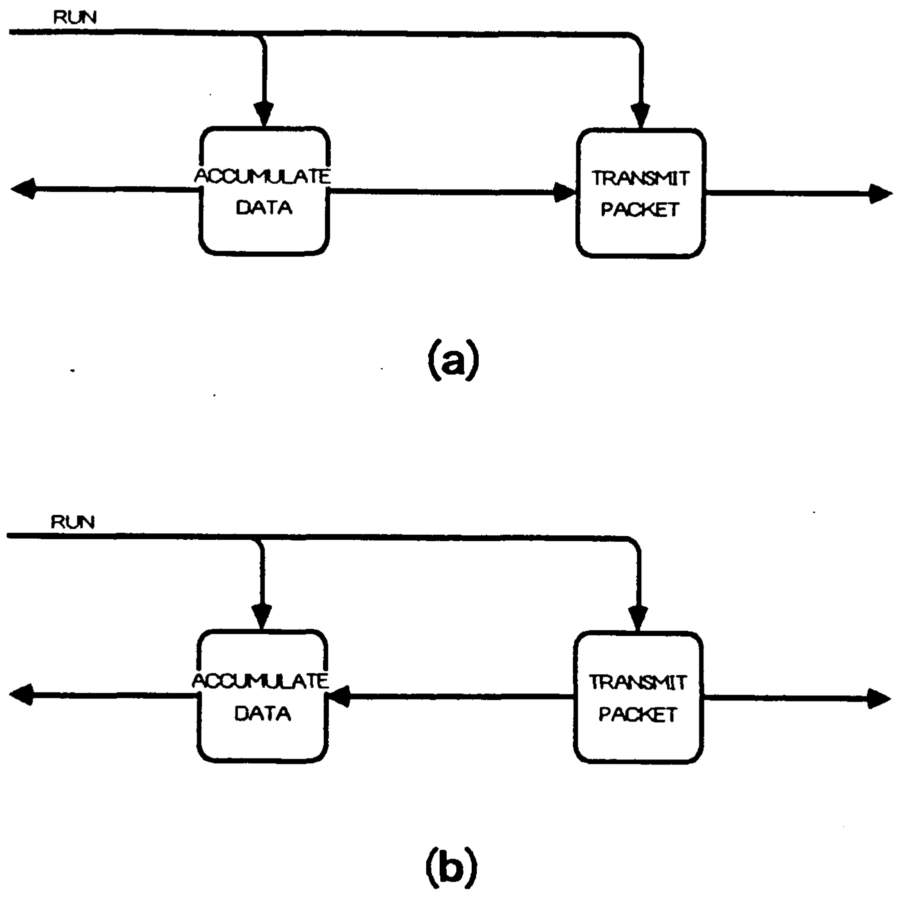


Figure 3-2. Concurrent Objects

3.2 DECOMPOSITION OF OBJECTS

At its top level, any complete system may be represented by a single object. For example, Figure 3-3 shows a diagram of the complete SCHEDULE ORGANIZER of the last section. The box labeled "USER" is an "external entity." An external entity is an object which is not included in the system, but which communicates with the top level system object. In this case terminal input/output operations are "serviced" by the USER. Note that this is a design diagram and thus shows the physical communications and data flows, not the higher level meaning that the data might have. Thus a user at a terminal sends and receives "TEXT" through the terminal operations.

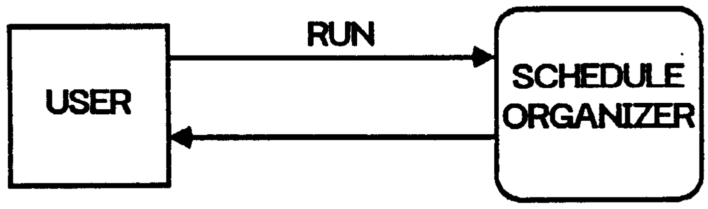


Figure 3-3. Schedule Organizer External Entities Diagram

A system level object may communicate with several external entities. A diagram such as Figure 3-3 showing these communications is an "external entities diagram." Communications with external entities are usually initiated by the system. In fact, external entities are often much like passive data objects, all of whose operations have a one directional data flow. A direct access or indexed file might be an exception to this, with a read operation taking an index as an argument and producing a record as the result. Another exception is that some external object must start the system. That is, initially control resides somewhere outside the system, and it must flow into the system for execution to begin. In Figure 3-3, the User invokes the SCHEDULE ORGANIZER using the RUN operation. A final example of control flowing into a system would be an asynchronous interrupt. This could be modeled by the interrupting entity invoking an operation in the concurrently running system. Servicing the operation would then model servicing the interrupt.

The object SCHEDULE ORGANIZER in Figure 3-3 represents a packaging of the complete object diagram of Figure 3-1. Working in the other direction, Figure 3-1 is a "decomposition" of the object SCHEDULE ORGANIZER. This can be expanded into the idea of stepwise refinement for objects and object diagrams. Beginning at the system level, each object can be refined into a lower level object diagram. The result is a leveled set of object diagrams which completely describe the structure of a system down to the procedural level.

For example, Figure 3-4 shows the decomposition of ADDRESS BOOK, which would be the beginning of the next level decomposition of Figure 3-1. The object descriptions for this diagram are:

```
ADD

     Provides:
          ADD (NAME + ADDRESS)

     Uses:
          FIND-ADDRESS
          ADDRESSES

CHANGE

     Provides:
          CHANGE (NAME + ADDRESS)

     Uses:
          FIND-ADDRESS
          ADDRESSES

LOOKUP

     Provides:
          LOOKUP (NAME) ADDRESS
```

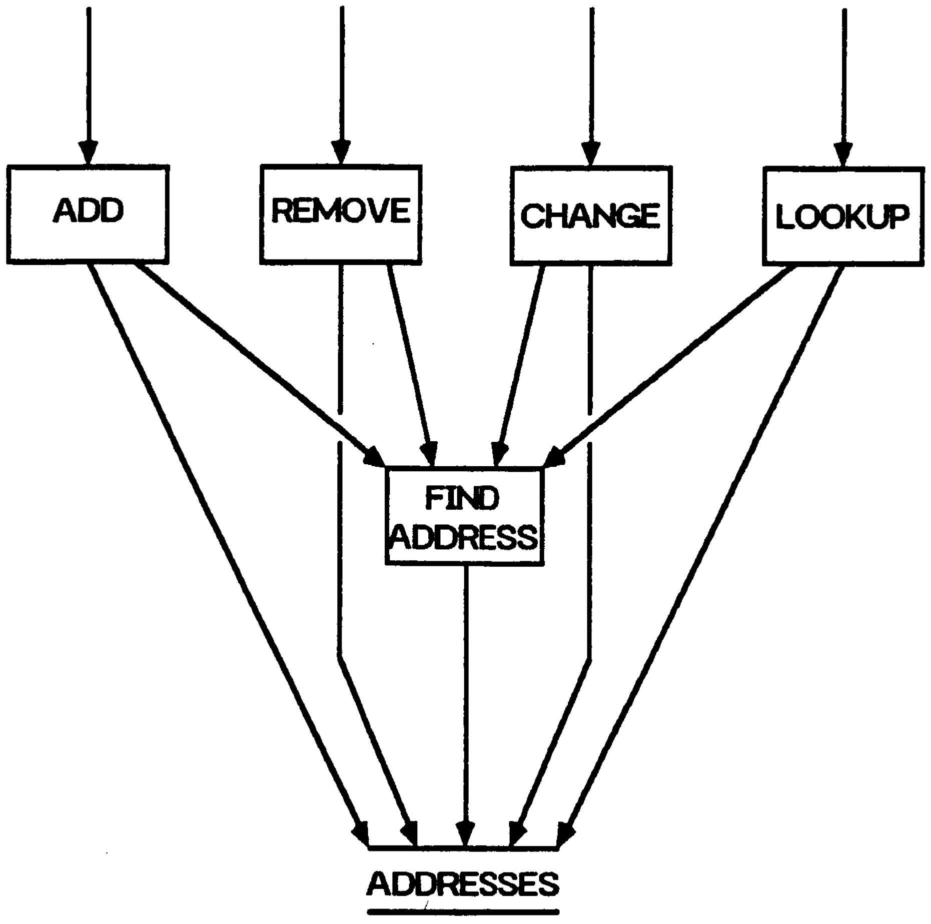


Figure 3-4. Address Book Decomposition

Uses:

FIND-ADDRESS

ADDRESSES

REMOVE

Provides:

REMOVE (NAME)

Uses:

FIND-ADDRESS

ADDRESSES

```
FIND-ADDRESS

     Provides:
          FIND-ADDRESS (NAME) INDEX

     Uses:
          ADDRESSES

ADDRESSES

     Contains:
          ADDRESS-LIST:  {NAME + ADDRESS}
```

All operations that lead "off the edges" of Figure 3-4 correspond to communications with the higher level ADDRESS BOOK object of Figure 3-1. This idea of "balance" is similar to that in leveled data flow diagramming. All operations provided by an object must appear in its decomposition diagram, and all communications "to the outside world" on the lower level diagram must be reflected in communications with the higher level object.

In Figure 3-4, the object ADDRESS BOOK has been completely decomposed into procedures. There is one procedure for each basic ADDRESS BOOK operation, and one additional procedure which is only used internally. Besides the procedures in Figure 3-4, there is also the object ADDRESSES. As in previous diagrams, an object such as this represents a store of data. Since it represents the internal state data of the higher level object, it is called a "state object." Procedures and states are really degenerate objects. Procedures are objects which have no internal state data and only service one operation. State objects contain data and only service operations to retrieve and update that data. All operations to a state object implicitly have one data flow into or out of the object. Note that the object description of ADDRESSES above indicates the this state object contains a list of names and their associated addresses.

Thus, using procedure and state objects, we have exposed the guts of ADDRESS BOOK as a state machine in the sense of Figure 2-4. At this low level we have defined exactly what state information and procedures are in ADDRESS BOOK. If necessary, it is now possible to further decompose the procedures by more traditional means. As a rule, procedures should not contain full objects or states. If they do, they should be considered as full objects themselves, even if they perform only one operation.

The main point of the above discussion is that any system can be represented as a single top-level object which can be successively decomposed, until at the lowest level we reach "degenerate objects." There are three types of degenerate objects. We have presented two types already: procedures and states. The third type is the "actor" object. Like a procedure, an actor has no state data. However, an actor does have state, in a sense, having to do with how it handles the flow of control. An actor object can control the servicing of its operations. This is primarily important in the communication between concurrent objects.

3.3 ACTOR OBJECTS

When a procedure operation is invoked, the procedure services it immediately. If more than one object concurrently invokes the operation at the same time, then they are serviced concurrently. Thus, an object which is made up of just procedures and states has no control of the servicing of its operations. If several operations are invoked concurrently, they will be serviced concurrently, without any coordination between them. This can cause undefined simultaneous alteration to internal state data and other unpleasant results. The basic problem is that multiple flows of control enter the object and proceed through it independently of each other. This problem can be solved with the use of actor objects.

An actor object can dynamically decide when to service one of its operations. An object which invokes an actor operation must wait first for the actor to decide to service the operation, and then for the servicing to be completed. Only one invocation of a specific operation is serviced at a time, in a first come, first serve order for each operation. If, while servicing one operation, an actor decides to service another, then the servicing of the first operation is effectively suspended until the servicing of the second one is finished. Thus, several control flows can enter an actor, but only one can be <u>active</u> at any one time.

As a simple example, an actor object can be used to represent a version of the classical semaphore (see Figure 3-5):

```
SEMAPHORE

     Provides:

          WAIT ()
          SIGNAL ()
```

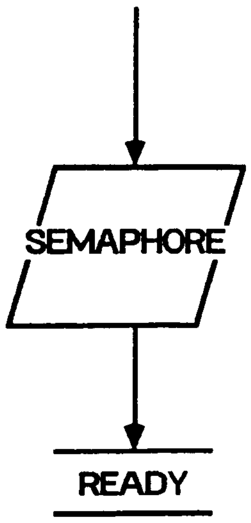


Figure 3-5. Semaphore Actor

```
READY

    Contains:

        READY : BOOLEAN
```

At any one time, the SEMAPHORE will service either the WAIT operation or the SIGNAL operation, but not both. It decides which operation to service by using a READY flag. If the SEMAPHORE is not READY, then it will accept only SIGNAL operations, and any objects invoking the WAIT operation will indeed have to wait. When the SEMAPHORE services a SIGNAL operation, it becomes READY. While the SEMAPHORE is READY, it will also service WAIT operations, of which there already may be some invocations pending. When it services a (single) WAIT operation, the SEMAPHORE once again becomes not READY until the next SIGNAL operation. We assume that initially the SEMAPHORE is READY. Note that there is no way to decompose SEMAPHORE into procedures, because the availability of its operations changes over time.

A semaphore can, for example, be used to ensure safe access to data common to concurrent objects. Figure 3-6 shows such a use. When either of the two procedures wants to access the common data, it invokes the SEMAPHORE WAIT operation. When this operation is serviced, the procedure can safely access or update the data, and all other accesses will be held up until the SEMAPHORE is SIGNALed. Alternatively, this same effect could be achieved by defining a new actor object (see Figure 3-7):

```
DATA-PROTECTOR

    Provides:

        READ () DATA
        WRITE (DATA)
```

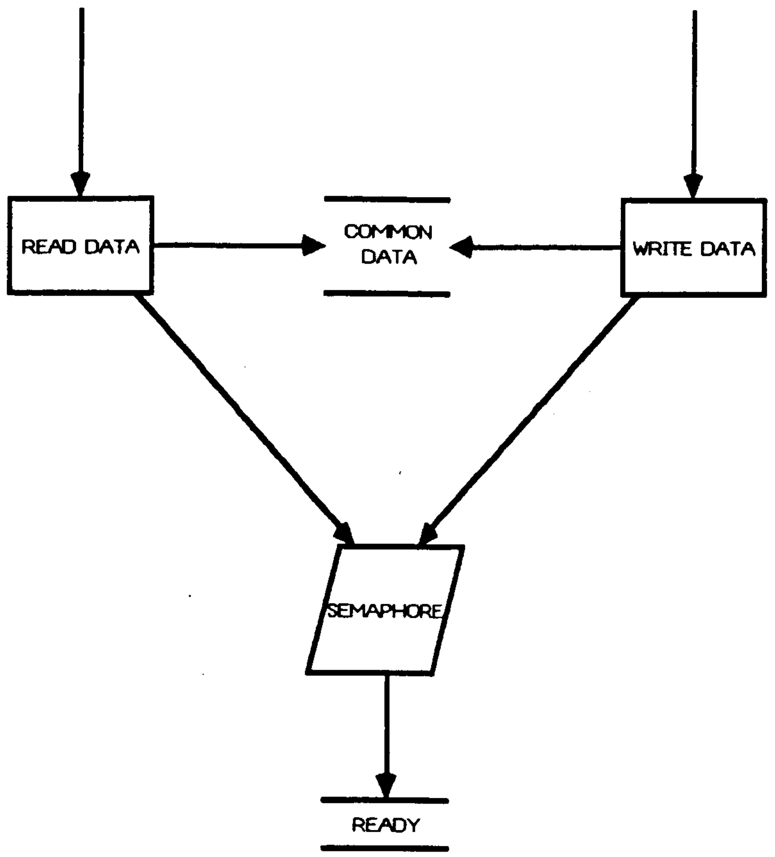


Figure 3-6. Use of a Semaphore

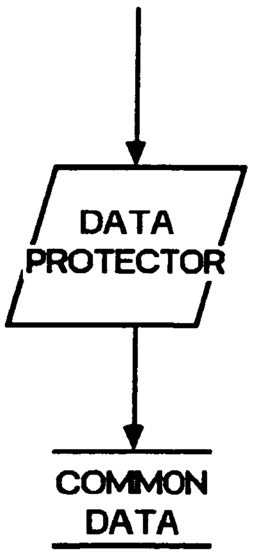


Figure 3-7. Data Protector Actor

The DATA PROTECTOR actor would only service one READ or WRITE operation at a time, thus protecting COMMON-DATA from simultaneous access. Either Figure 3-6 or 3-7 could be the decomposition of a "PROTECTED COMMON DATA" object which would provide READ and WRITE operations like the DATA PROTECTOR. However, in the composite object both the lower level use of actors and the internal state would be hidden.

3.4 TRANSLATING OBJECT DIAGRAMS INTO ADA

Using the object diagram notation, we can build a set of diagrams which completely describe the design structure of a system. Once this is done, the next step is to translate the design diagrams into code which provides a skeletal structure in which the remaining pieces of the system can be implemented. Though object diagrams provide a fairly general method for describing object-oriented designs, this translation step is most direct into Ada or similar languages. The correspondence between our object notation and Ada is straightforward:

Object Diagram	Ada
Object	Package
Procedure	Procedure/Function
State	Package/Task Variables
Actor	Entries/Accepts
Communication	Procedure/Function/Entry Call

To demonstrate the translation process, we return to the SCHEDULE ORGANIZER example. The first decomposition of this object was into three objects: USER INTERFACE, ADDRESS BOOK and DATE BOOK. We can now create package specifications for these objects based on the first level decomposition diagram

(Figure 3-1). Operations are defined in the package which services them. The resulting specifications are:

```
package USER_INTERFACE is
  procedure RUN;
end USER_INTERFACE;

package ADDRESS BOOK is
  type ADDRESS is
    record
      STREET    : STRING(1..30);
      CITY      : STRING(1..20);
      STATE     : STRING(1..2);
      ZIP       : STRING(1..5);
    end;

  procedure ADD
    (NAME: in STRING;
    ENTRY: in ADDRESS);
  procedure REMOVE
    (NAME: in STRING);
  procedure CHANGE
    (NAME: in STRING;
    ENTRY: in ADDRESS);
  function LOOKUP
    (NAME: in STRING)
    return ADDRESS;

end ADDRESS BOOK;

package DATE BOOK is

  type DATE is
    record
      YEAR      : INTEGER range 00 .. 99;
      MONTH     : INTEGER range 1 .. 12;
      DAY       : INTEGER range 1 .. 31;
    end record;

  type TIME is INTEGER range 0 .. 23;

  procedure GET_APPOINTMENT
    (DAY: in DATE;
    HOUR: in TIME;
    NAME: out STRING;
    PLACE: out ADDRESS_BOOK.ADDRESS);
  procedure MAKE_APPOINTMENT
    (DAY: in DATE;
    HOUR: in TIME;
    NAME: in STRING);
    procedure CANCEL_APPOINTMENT
      (DAY: in DATE;
      HOUR: in TIME);

  end DATE BOOK;
```

The main program would then have the form:

```
  procedure SCHEDULE ORGANIZER is
    -- global type definitions
        .
        .
        .
    -- package specifications
        .
        .
        .
    package body USER_INTERFACE is separate;
    package body ADDRESS_BOOK is separate;
    package body DATE_BOOK is separate;

  begin
    USER_INTERFACE.RUN;
  end SCHEDULE_ORGANIZER;
```

The system RUN operation in Figure 3-3 represents the invocation of the SCHEDULE_ORGANIZER main procedure by the user. This in turn causes the call of USER_INTERFACE.RUN, passing the flow of control to the USER_INTERFACE. Note that package USER_INTERFACE has only the one RUN procedure. Since it has only this one operation and since it is active for the entire time the system is running, it would be acceptable to implement USER_INTERFACE as a procedure. The main program would then be just the call "USER_INTERFACE". Note that this would not change the status of USER INTERFACE as an object on the object diagram (Figure 3-1). At the next level, we could now code the declarative part of the bodies of the above three packages. As an example, consider the ADDRESS_BOOK package. From the decomposition object diagram for object ADDRESS BOOK (Figure 3-4), we can construct the following body:

```
separate (SCHEDULE_ORGANIZER)
package body ADDRESS_BOOK is
  -- type definitions
  type ADDRESS_RECORD is
    record
      NAME          : STRING;
      ENTRY         : ADDRESS;
    end record;
  BOOK_SIZE         : constant := 100;
  type ADDRESS LIST TYPE is
    array (1..BOOK_SIZE) of ADDRESS_RECORD;
  -- internal state
  ADDRESSES_LIST    : ADDRESS_LIST_TYPE;
        .
        .
        .
  procedure ADD (NAME: in STRING; ENTRY: in ADDRESS) is
  separate;
  procedure REMOVE (NAME: in STRING) is separate;
  procedure CHANGE (NAME: in STRING; ENTRY: in ADDRESS)
  is separate;
  function LOOKUP (NAME: in STRING) return ADDRESS is
  separate;
end ADDRESS_BOOK;
```

To complete the system, we could implement the remaining procedures using more traditional functional design methods.

The only applicable Ada unit not used in the above example is the task. In Ada, for the flow of control to actually reside in two units at the same time, these units must be tasks. Therefore, if there are concurrent objects in an object diagram, then at least some part of them must be translated into tasks. Actually, higher level concurrent objects which are decomposed into other objects generally can still be translated as just packages. At the lowest level, however, at least some of the degenerate objects composing the higher level object must be tasks. The degenerate object that usually signals the use of an Ada task is the actor.

An actor represents quite closely the rendezvous mechanism of an Ada task body. An Ada task can, however, have internal state data, while an actor cannot. Thus an actor would translate into a task without a declarative part, and any state data would be contained in a surrounding package. It is common to combine the surrounding package with the task to create a composite Ada object which represents the actor and the data that it alone uses. For example, the SEMAPHORE actor of Figure 3-5 could be translated into:

```
task SEMAPHORE is
  entry SIGNAL;
  entry WAIT;
end SEMAPHORE;

task body SEMAPHORE is
  READY   : BOOLEAN := TRUE;  -- state object READY

begin
  loop                        -- begin actor SEMAPHORE
    select
      when READY =>
      accept WAIT;
      READY := FALSE;
    or
      accept SIGNAL;
      READY := TRUE;
    else
      terminate;
    end select;
  end loop;                   -- end of actor
end SEMAPHORE;
```

Note how the READY flag is included in the task, and how the actor object represents the executable part of the task body. A rendezvous with the task corresponds to a communication with the actor object and accepting an entry corresponds to servicing an operation.

Now, if we used SEMAPHORE to implement a PROTECTED COMMON DATA object with the decomposition shown in Figure 3-6, we could translate the object as a package even though it would be concurrent with other objects. It would, however, contain the SEMAPHORE task as the translation of part of its decomposition. The higher level package translation might be:

```
package PROTECTED_COMMON_DATA is
  procedure READ(X: out DATA);
  procedure WRITE(X: in DATA);
end PROTECTED_COMMON_DATA;

package body PROTECTED_COMMON_DATA is

  COMMON_DATA           : DATA;      -- internal state

  task SEMAPHORE is
    entry SIGNAL;
    entry WAIT;
  end SEMAPHORE;

  task body SEMAPHORE is separate;

  procedure READ(X: out DATA) is
  begin
    SEMAPHORE.WAIT;
    X := COMMON_DATA;
    SEMAPHORE.SIGNAL;
  end READ;

  procedure WRITE(X: in DATA) is
  begin
    SEMAPHORE.WAIT;
    COMMON_DATA := X;
    SEMAPHORE.SIGNAL;
  end WRITE;

end PROTECTED_COMMON_DATA;
```

The design of Figure 3-7 would actually be a better use of Ada tasking. In this case, PROTECTED COMMON DATA could be translated into a single task:

```
task PROTECTED_COMMON_DATA is
  entry READ (X: out DATA);
  entry WRITE (X: in DATA);
end PROTECTED_COMMON_DATA;

task body PROTECTED_COMMON_DATA is

  COMMON_DATA            : DATA;      -- internal state

begin
  loop                          -- begin actor DATA-PROTECTOR
    select
      accept READ (X: out DATA) do
        X := COMMON_DATA;
      end READ;
    or
      accept WRITE (X: in DATA) do
        COMMON_DATA := X;
      end WRITE;
    or
      terminate;
    end select;
  end loop;                     -- end of actor
end PROTECTED_COMMON_DATA;
```

SECTION 4 - OBJECT-ORIENTED DESIGN

Using the concepts and notation of object diagrams, this section deals with two main questions:

- What makes a good object?
- How are designs constructed from objects?

While we cannot provide all-encompassing answers to these questions, we do provide principles to guide the design process. They are heuristics, not laws, but they do provide a powerful means for constructing and comparing alternative designs. They are thus tools to aid the software designer in his (or her) engineering art.

4.1 PRINCIPLES FOR DESIGNING OBJECTS

The intent of an object is to represent a problem-domain entity. The concept of "abstraction" deals with how an object presents this representation to other objects [Dijkstra 68, Liskov 74, Ledgard 77, Booch 83]. As software models, objects should also act as black boxes to allow easy debugging and maintenance. The concept of "information hiding" deals with what an object keeps secret from other objects [Parnas 72]. These two concepts provide the main guides for assessing an object. A "good" object thus represents a problem domain entity and hides closely-related information that is likely to change if the implementation of the object changes.

There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence. The following are some points on that scale:

Entity Abstraction	Best
Action Abstraction	↓
Virtual Machine Abstraction	
Coincidental "Abstraction"	Worst

Each kind of abstraction in this scale is a subset of the kind below it.

An "entity abstraction" is an object which represents a useful model of a problem domain entity. The entity could be as concrete as a hardware sensing device or more abstract, such as a compiler symbol table. We include "data abstraction" under entity abstraction as denoting objects which define type managers.

"Action abstraction" moves from abstracting the properties of <u>things</u> to abstracting the properties of <u>actions</u>. An action abstraction is an object which provides a generalized set of operations which all perform the same kind of action. A general "input handler" or a "math processor" would be action abstractions. Procedures are generally action abstractions.

"Virtual machine abstractions" are objects which group together operations which are all used by some superior level of control or all use some junior level set of operations. While the concept of a "virtual machine" will be useful later on, it is not a very good criterion for constructing objects. Such objects group together unrelated actions on the basis of their being at about the same "level of control."

Finally, "coincidental abstraction" is really no abstraction at all. A coincidentally abstract object packages a set of operations which have no relation to each other in any substantial way and probably do not even get along well together.

Information hiding is complementary to abstraction. The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of information hiding states that such details should be kept secret from other objects [Parnas 72, Booch 83]. While good abstraction promotes information hiding, and often vice versa, it is possible to construct objects which have high abstraction, but provide ways to expose their contents. Conversely, it is possible to hide information well without constructing good abstractions. The best objects should thus be constructed to provide operations on abstract entities and to carefully hide internal representations and related secrets.

4.2 PRINCIPLES FOR DESIGNING SYSTEMS

Following [Rajlich 85], we will consider two basic orthogonal hierarchies in software system designs. The "parent-child hierarchy" deals with the decomposition of larger objects into smaller component objects (as discussed in Section 3.2). The "seniority hierarchy" deals with the organization of a set of objects into "layers." Each layer defines a "virtual machine" [Dijkstra 68] which provides a set of services to senior layers.

The object diagram notation can distinctly represent these hierarchies. The leveling of object diagrams directly expresses the parent-child hierarchy (see Figure 4-1). On the other hand, the topology of connections on a single object diagram shows the seniority hierarchy (see Figure 4-2). (Note the quite literal orthogonality of these two hierarchies in Figure 4-1!) Any layer in a seniority hierarchy can call on any operations in junior layers, but never any operation in a senior layer. Thus, if we group objects into virtual machine layers, these layers are always related by a directed, acyclic graph. From Figure 4-2 we would get the graph shown in Figure 4-3. All cyclic relationships between objects must be contained within a virtual machine layer.

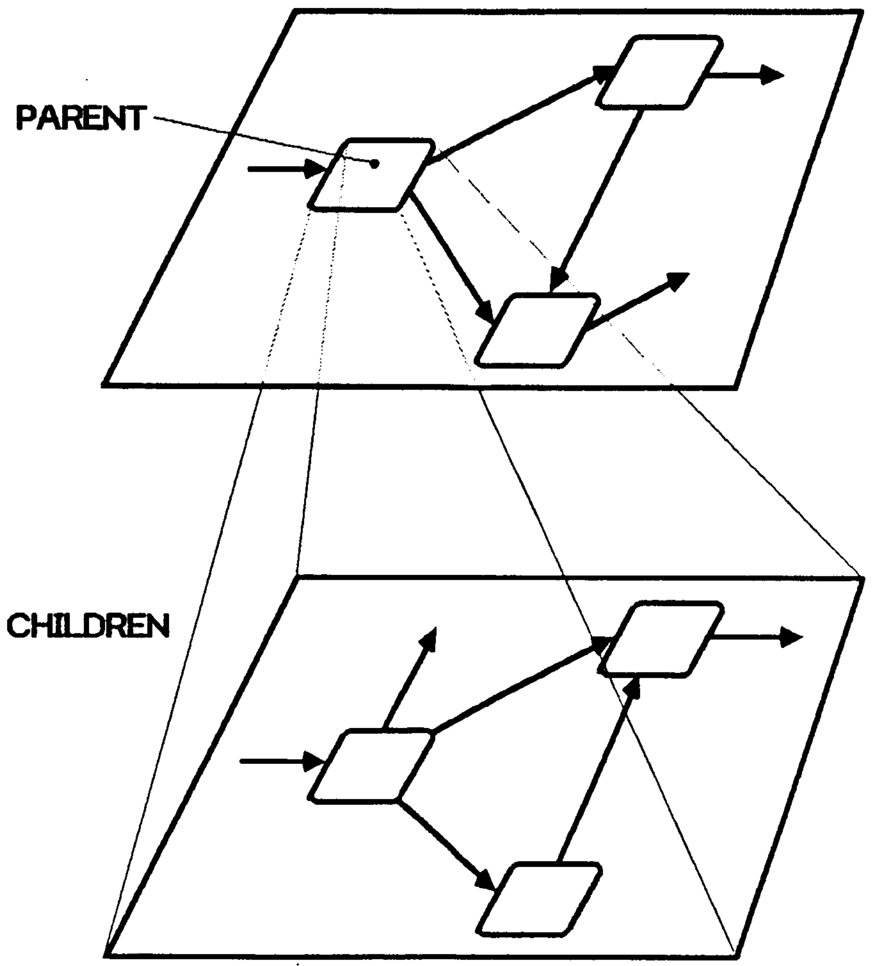


Figure 4-1. Parent-Child Hierarchy

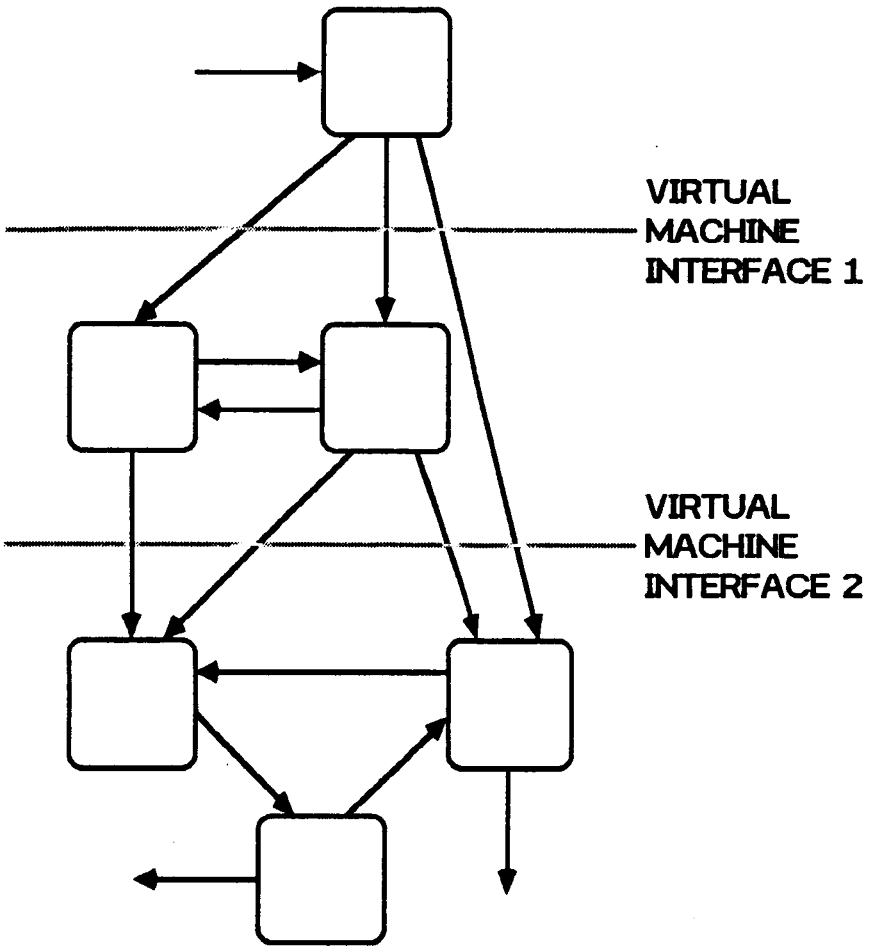


Figure 4-2. Seniority Hierarchy

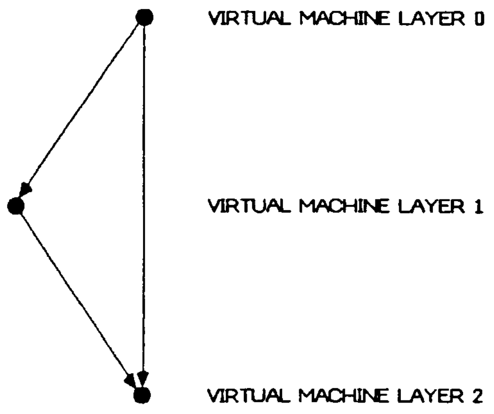


Figure 4-3. Virtual Machine Graph

The general structure of an object-oriented design as presented here is a seniority hierarchy of virtual machines, each of whose components is decomposed into children objects. The children of each object are themselves organized in seniority hierarchies, and so on. Figure 4-4 shows a stylized overview of the top level object diagram of such a system. Figure 4-4 uses the words "afferent" and "efferent" in the input/output sense of [Yourdon 79]. Thus the virtual machines provide operations for THE SYSTEM to input, process and output data. Lower level object diagrams will also have a structure similar to Figure 4-4. However, instead of a single most-senior object, they will generally have a set of senior level objects which implement the operations of the parent object. These senior objects use the junior virtual machine operations to do this.

Note that we have not made the virtual machine layers in Figure 4-4 into objects themselves. Such objects would generally have only (surprise!) virtual machine abstraction. Each virtual machine layer should therefore be composed of objects with higher abstraction. Figure 4-5 shows one approach, reminiscent of structured design [Yourdon 79]. These virtual machine components would have, at best, action abstraction. A better approach is to identify appropriate problem domain entities and create entity abstractions which package afferent, transform and efferent operations for each entity (see Figure 4-6). The parent-child decomposition of the most-senior object THE SYSTEM might still be a structured design style afferent-transform-efferent hierarchy. But now it could be designed as if the virtual machine operations where "primitive operations" in an extended language. These "junior level" (in a control sense) operations are themselves defined within objects which represent the specific entities with which the operations deal.

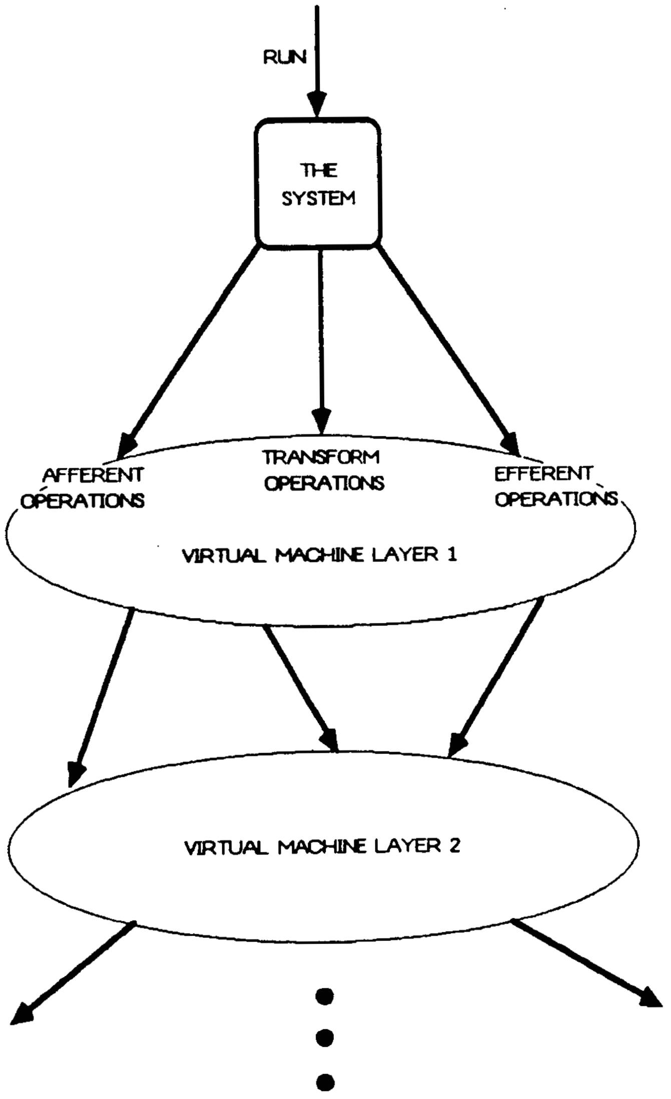


Figure 4-4. Top-Level Object Diagram

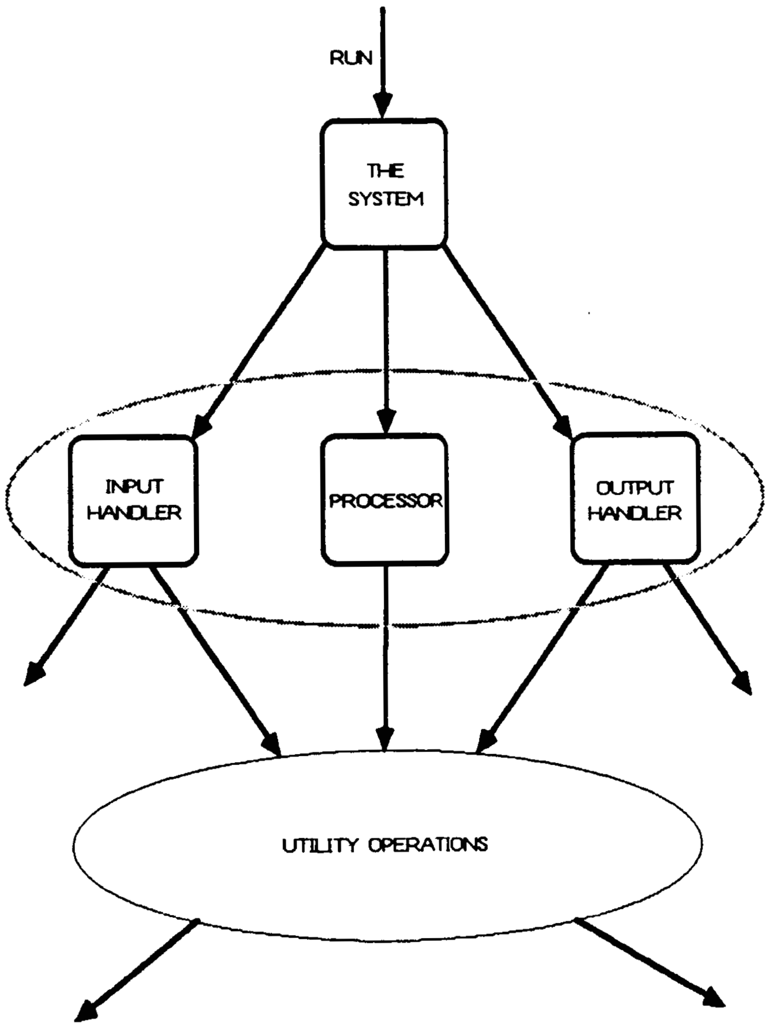


Figure 4-5. Input/Process/Output Virtual Machine

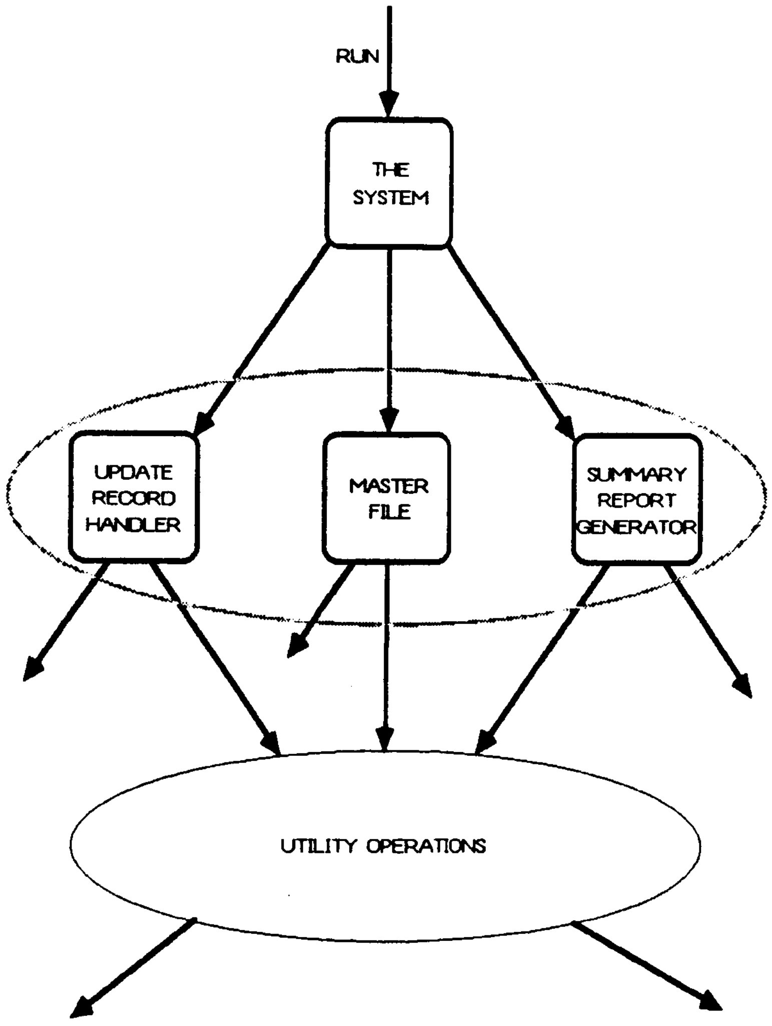


Figure 4-6. A Better Virtual Machine Decomposition

The seniority hierarchy deals mainly with control: the senior levels control the operation of the junior levels. To varying degrees, senior levels can also control the data flow and interaction between components of junior layers. Consider the automated manufacturing plant simulation system diagrammed in Figure 4-7. Note that the junior components do not interact directly. As part of its use of the virtual machine operations, the PLANT SIMULATOR must control the flow of data between the three virtual machine components. This has the advantage that none of the junior components needs to know anything about any of the other components. However, the senior object has to do a lot of work simply passing data from one junior object to another.

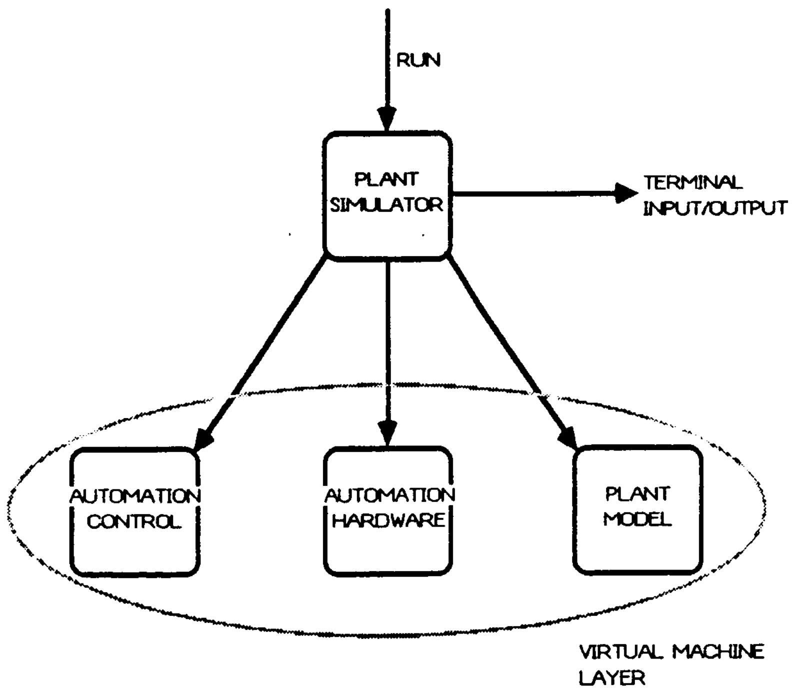


Figure 4-7. An Automated Manufacturing Plant Simulation System

Suppose we remove the data flow control from the senior object and let the junior objects pass data directly (see

Figure 4-8). This type of design was, in fact, used for part of our pilot project simulator. The senior object has been reduced to simply activating various operations in the virtual machine. These operations can then use other operations internal to the virtual machine to pass data and commands between component objects. This means that some objects must have knowledge of some other objects within the virtual machine layer, limiting any possible future uses of the components apart from this virtual machine. An added complication is the possible need for buffering of incoming data as state information in some objects, until the next control activation from the senior object.

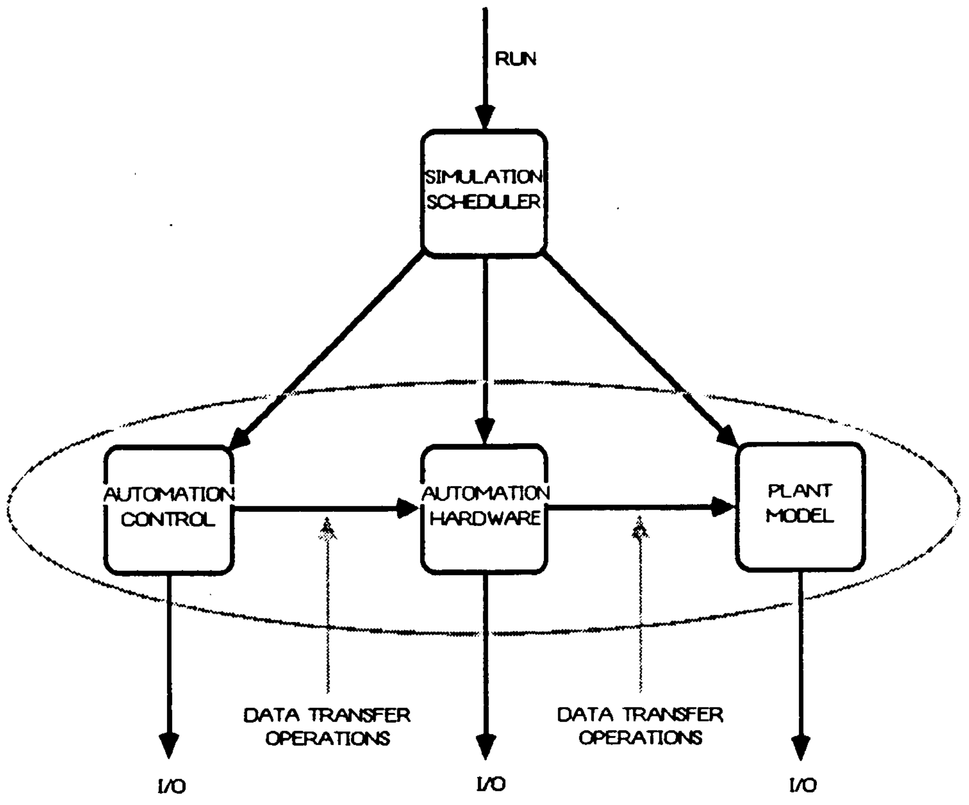


Figure 4-8. Plant Simulator With Junior-Level Connections

We can even remove the senior object completely by distributing control among the junior objects. By making the remaining objects concurrent and passing data through synchronizing rendezvous, we can also often eliminate the above need for buffering. Figure 4-9 shows an example of such a design. The seniority hierarchy has collapsed, leaving a "homologous" or non-hierarchical design [Yourdon 79] (non-seniority-hierarchical, that is; the parent-child hierarchy still remains). A design which is homologous at all parent-child levels is very similar to what would be produced by George Cherry's PAMELA[1] methodology for real-time applications [Cherry 85a, Cherry 85b].

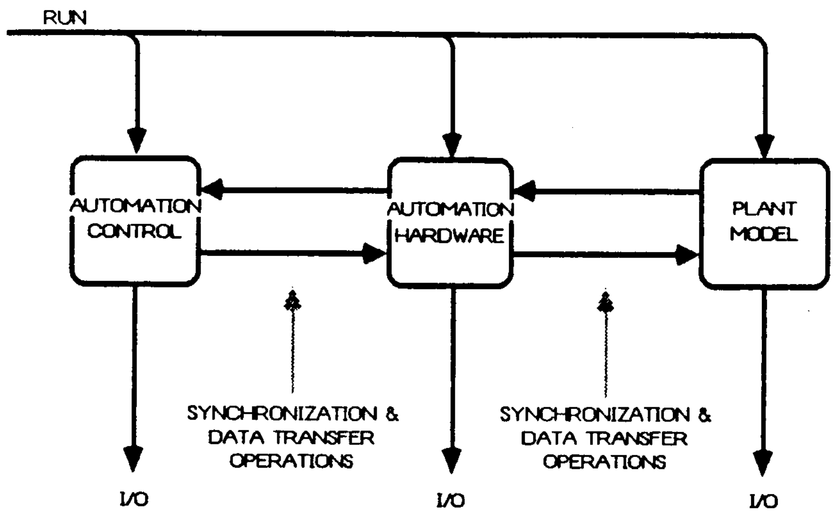

RUN
AUTOMATION CONTROL
AUTOMATION HARDWARE
PLANT MODEL
SYNCHRONIZATION & DATA TRANSFER OPERATIONS
SYNCHRONIZATION & DATA TRANSFER OPERATIONS
I/O
I/O
I/O

Figure 4-9. Plant Simulator, Homologous Design

It is sometimes possible to recover a seniority hierarchy from a seemingly homologous design. Figure 4-10 shows a simplified real-time aircraft on-board monitoring system. The system is highly concurrent without centralized control. Note the representation of interrupts ("SMOKE ALARM" and "KEYSTROKE") as operations originating outside the system. Due to the real-time nature of the system, actions of the system are caused by outside events, with control flowing,

[1]PAMELA is a trademark of George W. Cherry.

roughly, from the left in Figure 4-10 to the right, where results are displayed for the pilot. Thus, by turning Figure 4-10 on its side, the objects become organized in a seniority hierarchy (see Figure 4-11). The diagram has been reorganized according to calling directions. Since action is initiated by external stimulus, the input interfaces are at the top of the seniority hierarchy as two concurrent, most senior objects. The system is <u>input driven</u> in a very literal sense: the senior-level, controlling objects are the ones closest to the input. The junior-level, controlled objects produce the output. This is a quite natural organization for such an embedded, real-time system.

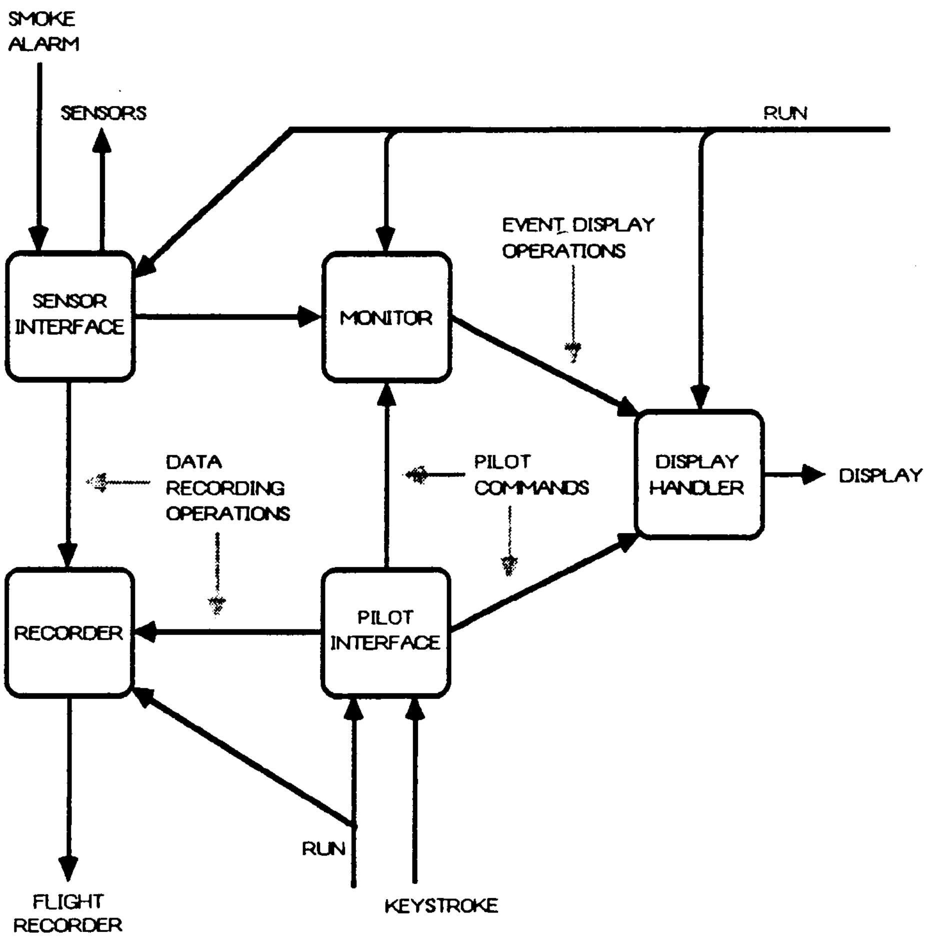


Figure 4-10. Aircraft Monitoring System

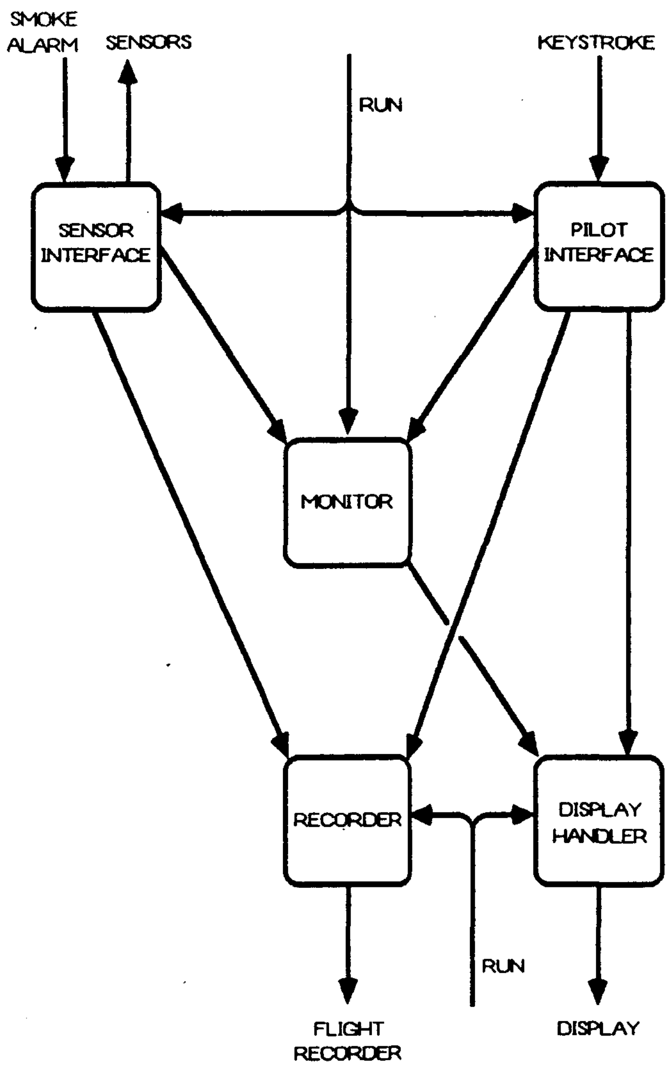


Figure 4-11. Aircraft Monitoring System With Seniority Hierarchy

The main advantage of a seniority hierarchical design is that it reduces the "coupling" (in the sense of [Yourdon 79]) of the virtual machine components. This is because each virtual machine layer needs to know nothing about its seniors. It is possible to completely replace senior-level controllers without affecting the junior levels at all. In the stronger version where the senior levels also control data flow, the virtual machine component objects are even decoupled from each other. This means that they are particularly adaptable to future use, and that they are less likely to propagate or be affected by changes in the system.

The centralization of the procedural and data flow control can make the system easier to understand and modify. On the other hand, this very centralization can cause a messy bottleneck in the data flow between objects. Even if this is eliminated, complicated scheduling can sometimes result in a similar control bottleneck. In addition, if the control and scheduling of junior objects depends heavily on information internal to them, then centalizing control could reduce their level of information hiding and abstraction. In this case a more homologous design would be appropriate. In large real-time systems with low level external stimuli, it can be particularly useful to eliminate the senior level data and control bottleneck and take advantage of distributed, concurrent control [Cherry 85a]. Even in this case it is sometimes possible, as discussed above, to recast a concurrent, homologous design in the form of a seniority hierarchy without the usual disadvantages. In general, however, the best design will be between the extremes of use of the seniority hierarchy.

SECTION 5 - ABSTRACTION ANALYSIS

Object diagrams and the object-oriented design concepts discussed in the previous sections can be used as part of an object-oriented life cycle. Section 3.4 described how object diagrams can be translated into Ada. However, we must also be able to create an initial object-oriented design from a system specification. We use structured analysis to develop the specification [DeMarco 79]. The data flow diagrams of a structured specification provide a leveled, graphical notation containing the information needed to represent abstract entities, but in a form emphasizing data flow and data transformation. "Abstraction analysis" is the process of making the transition from a structured specification to to an object-oriented design [Stark 86].

The main idea in producing an initial design is to identify objects, map them back to the requirements, and then identify the operations. Abstraction analysis transforms a structured specification into an object-oriented design by first identifying abstract entities and a tentative control hierarchy, and then identifying objects, operations, and a hierarchy of virtual machines. As an intermediate step between data flow diagrams and the control-flow oriented object diagrams we create an "entity graph". This graph shows the interconnections of the abstract entities in the problem domain from a control point of view, where the data flow diagrams give a data exchange point of view. Since the direction of control and design complexity are also considered in creating an object diagram, the best objects and the best abstract entities are not necessarily the same.

Operations are identified from processes and data stores contained by an object, and by the data flow between objects. Fortunately, data flow diagrams are analogous to object diagrams in that they are developed from a higher level of abstraction to a more detailed view. Our approach can generate leveled object diagrams because of this property. We will first discuss the ideas used in performing abstraction analysis and how they are used to identify objects. We will then discuss the entire process of designing from a structured specification.

Section 4 discussed the tradeoff between the loose coupling generated by a strong seniority hierarchy and the real time performance of a homologous design. Here we will describe the nature of abstraction and control issues that have to be faced. The procedure used to produce an object diagram first entails identifying central entities and virtual interfaces, secondly identifying objects, and then using the results of these two steps to produce an object diagram.

We will illustrate this process with a version of the Gamma Ray Observatory (GRO) Attitude Dynamics Simulator (GRODY) pilot project [Agresti 86]. Analysts use an attitude dynamics simulator to verify the correctness of a spacecraft's attitude control laws. Such a system must simulate the spacecraft control system, model the spacecraft's response to control and provide simulated input to the control system. Figures 5-1 and 5-2 are the two highest level data flow diagrams used in this example.

5.1 IDENTIFYING CENTRAL ENTITIES AND VIRTUAL INTERFACES

A "central entity" in abstraction analysis is nearly identical to a central transform in structured design. In structured design [Yourdon 79] input and output data flows are examined and followed inwards until they reach the highest level of abstraction. The processes between the inputs and the outputs form the central transform. In abstraction analysis a designer does the same, but also examines the central transform to determine which processes and states

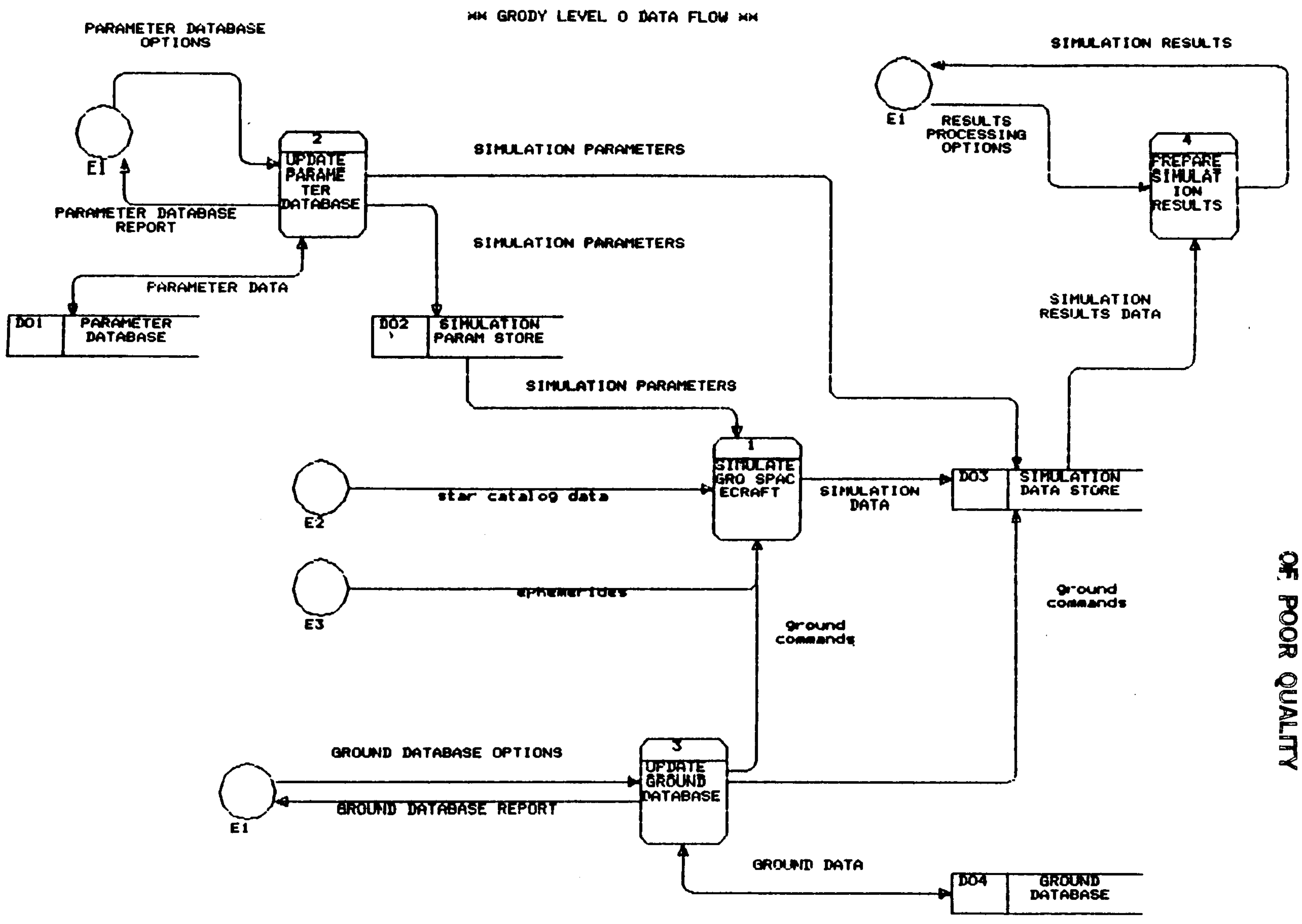


Figure 5-1. GRODY Level 0 Data Flow Diagram

ORIGINAL PAGE IS OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

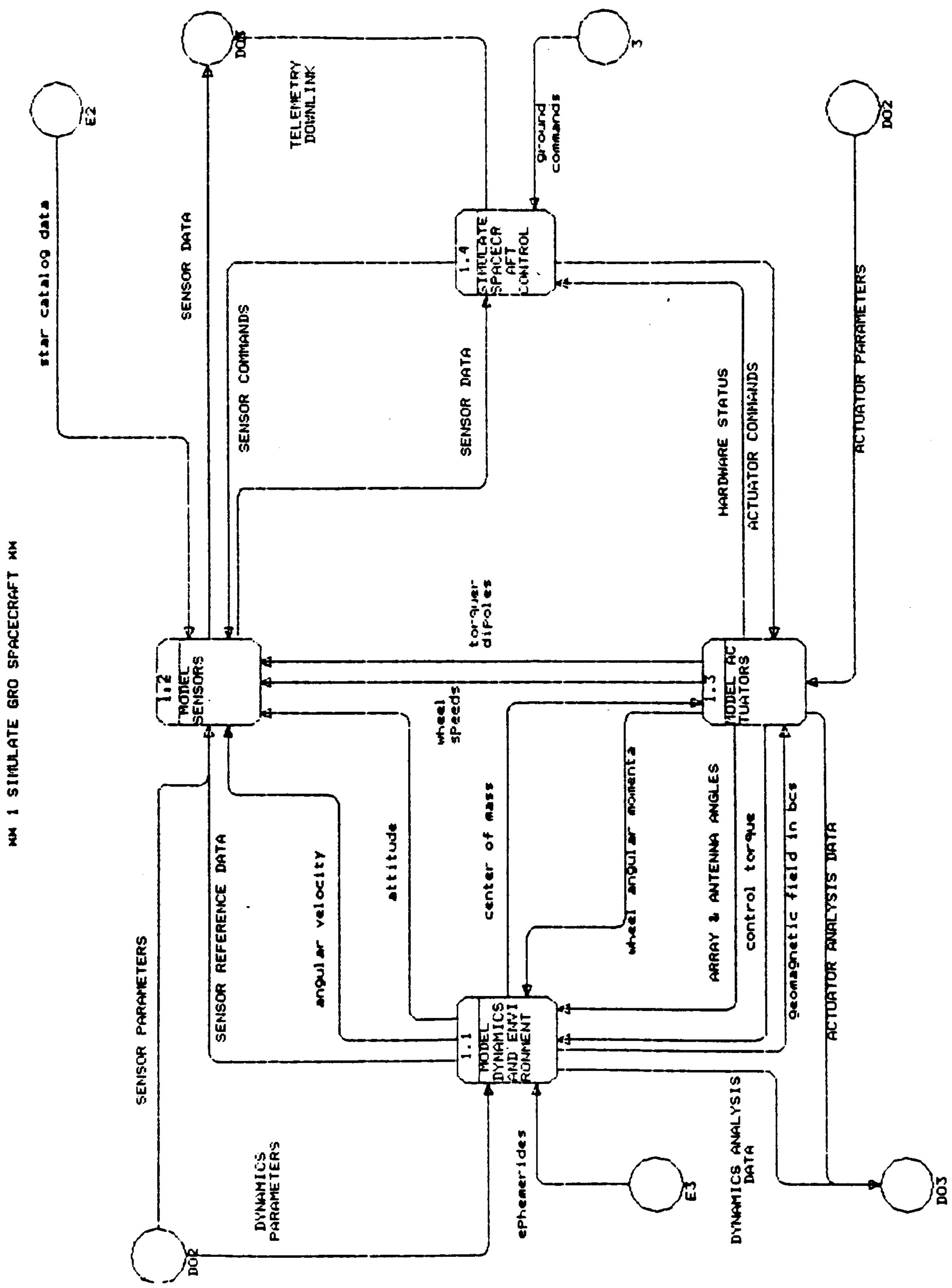


Figure 5-2. 1. Simulate GRO Spacecraft

represent the best abstract model of what the system does. For example, it is clear from Fiqure 5-1 that SIMULATE GRO SPACECRAFT is the central transform for GRODY. Examininq Fiqure 5-2 we can arque that SIMULATE SPACECRAFT CONTROL is the central entity, as the purpose of a dynamics simulator is to test the control laws. We could continue to desiqn usinq either assumption. We choose to make SIMULATE SPACECRAFT CONTROL the central entity.

After identifyinq the central entity we identify what abstract entities are supportinq it. The idea is to follow the afferent and efferent data flows away from the central entity and to qroup related processes and states alonq these data flows, forminq abstract entities.

In creatinq the level 0 object diaqram for GRODY we will build a recast data flow diaqram step by step as we identify abstract entities. Fiqure 5-3 shows the process SIMULATE SPACECRAFT CONTROL and the adjacent processes and data flows. We look at the data flows in and out of the central entity and identify entities supportinq these data flows. This is done by qroupinq these processes and states into entities with hiqh abstraction. With GRODY, each process and state in Fiqure 5-3 maps to an entity. This is due to the specification beinq hiqhly abstract at the top level, rather than to any rule mappinq data flow diaqram processes directly into entities. Later examples show how related processes are grouped into a sinqle entity. Groupinq related processes may require examininq lower level data flow diaqrams, although in this case it does not.

To ensure that we start with a stronq seniority hierarchy we use the concept of virtual machine layers discussed in Section 4.2. We start by assuminq the existence of a "most-senior" object that calls on a virtual machine consistinq of the central entity and the entities that directly support the central entity. Fiqure 5-4 is an entity qraph for

ORIGINAL PAGE IS
OF POOR QUALITY

GRODY that contains only the highest virtual machine level. Squares represent entities, with the identifying numbers of processes or states from the data flow diagram written in the squares to show the mapping between requirements and entities. Arrows show the flow of control between GRODY and the virtual machine entities, and lines with no indication of direction represent potential communications between entities.

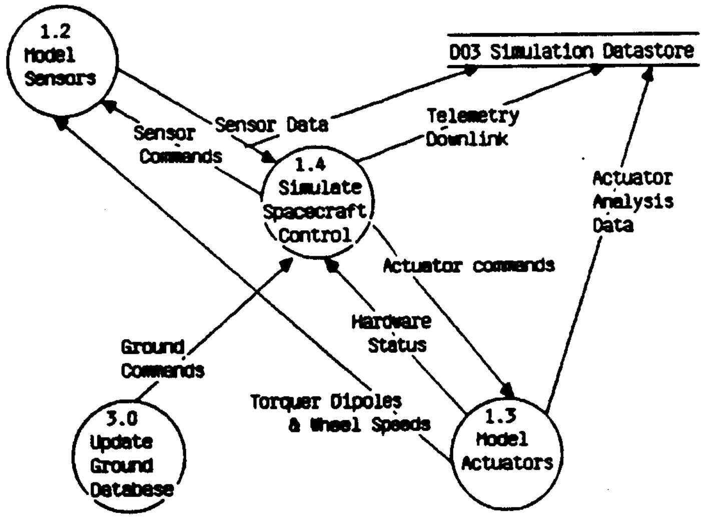
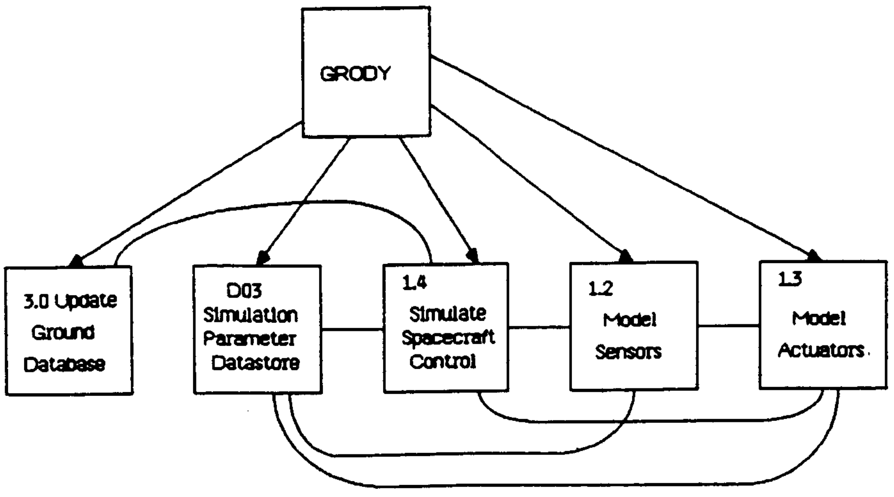


Figure 5-3. Support of Central Entity



GRODY
3.0 Update Ground Database
D03 Simulation Parameter Datastore
1.4 Simulate Spacecraft Control
1.2 Model Sensors
1.3 Model Actuators

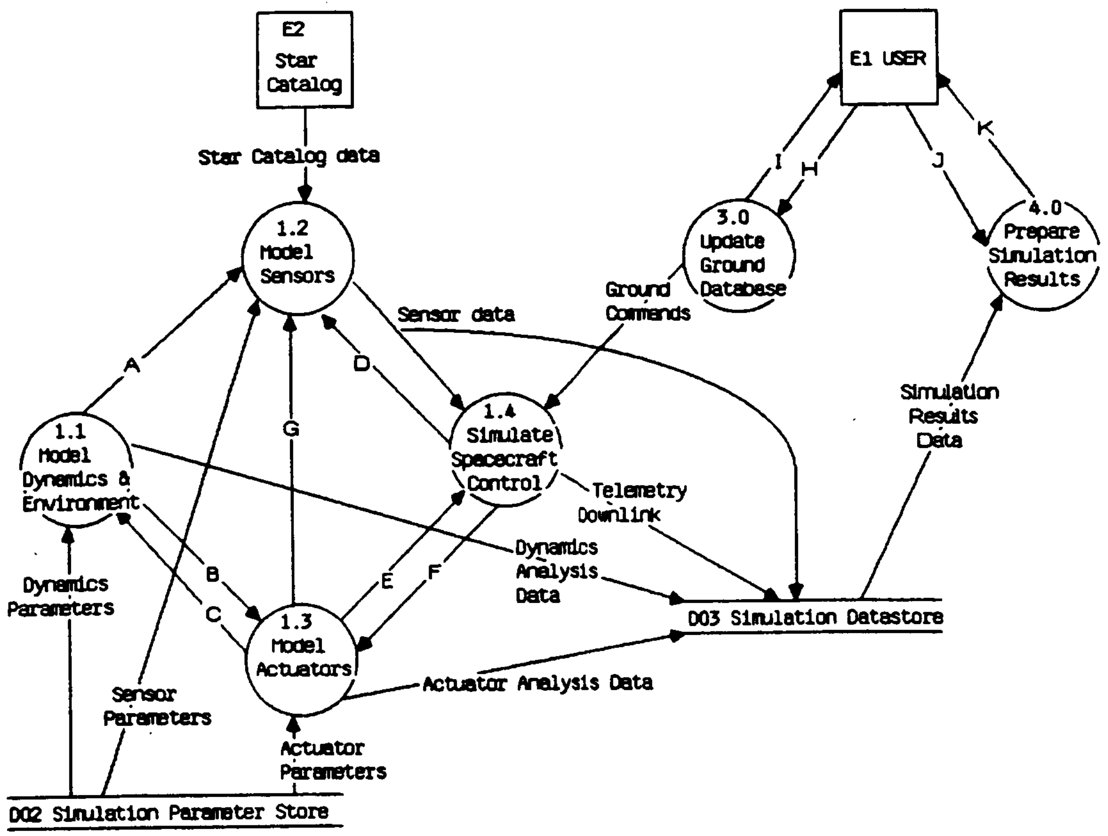
Figure 5-4. First Level of Entities

At this point the only control flow we want to see is the "most senior" object controlling the first group of entities identified. We add other control flows later, first in response to system requirements not captured on data flow diagrams, and secondly to optimize our virtual machine hierarchy. We have not made any determination about how data are passed between the entities. The options are to pass data directly between entities or to pass it through the most-senior entity.

The next group of entities is again identified by examining data flows, processes and states; this time the ones that are one step further removed from the central entity. For example, Figure 5-2 shows that the processes MODEL SENSORS and MODEL ACTUATORS are both supported by 1.1 MODEL DYNAMICS & ENVIRONMENT and the data store D02 SIMULATION PARAMETER STORE, and by D03 SIMULATION DATASTORE. MODEL SENSORS is also supported by the external entity STAR CATALOG. Similarly, Figure 5-1 shows that UPDATE GROUND DATABASE is supported by the user, and that the SIMULATION DATASTORE is supported by PREPARE SIMULATION RESULTS which is supported by the user. We then draw a data flow diagram (Figure 5-5) reflecting these relationships. To make identifying objects easier, we leave the names of processes and data stores on the diagram, but use shorthand labels for data flows when needed. In Figure 5-5 we have used shorthand labels in areas where the interactions are more complex.

Identifying entities is almost as straightfoward for this part of GRODY as it was for the first set of entities. The user is already an external entity. PREPARE SIMULATION RESULTS and MODEL DYNAMICS & ENVIRONMENT also map directly into entities. By examining the data flows coming from the datastore SIMULATION PARAMETER STORE we can determine that

ORIGINAL PAGE IS
OF POOR QUALITY



Dataflow labels

A: Sensor Reference Data • Attitude • Angular Velocity
B: Center of Mass • Geomagnetic Field in BCS
C: Wheel Angular Momenta • Array & Antenna Angles
D: Sensor Commands
E: Hardware Status
F: Actuator Commands
G: Wheel Speeds • Torquer Dipole
H: Ground Database Options
I: Ground Database Report
J: Results Processing Options
K: Simulation Results

Figure 5-5. DFD With Added Processes

this data store can be separated into parts supporting processes 1.1, 1.2, and 1.3. Thus the initial conditions for these three processes are associated with the appropriate entity, rather than having a separate entity acting as a global data area. Figure 5-6 shows the entity graph with the newly identified entities added.

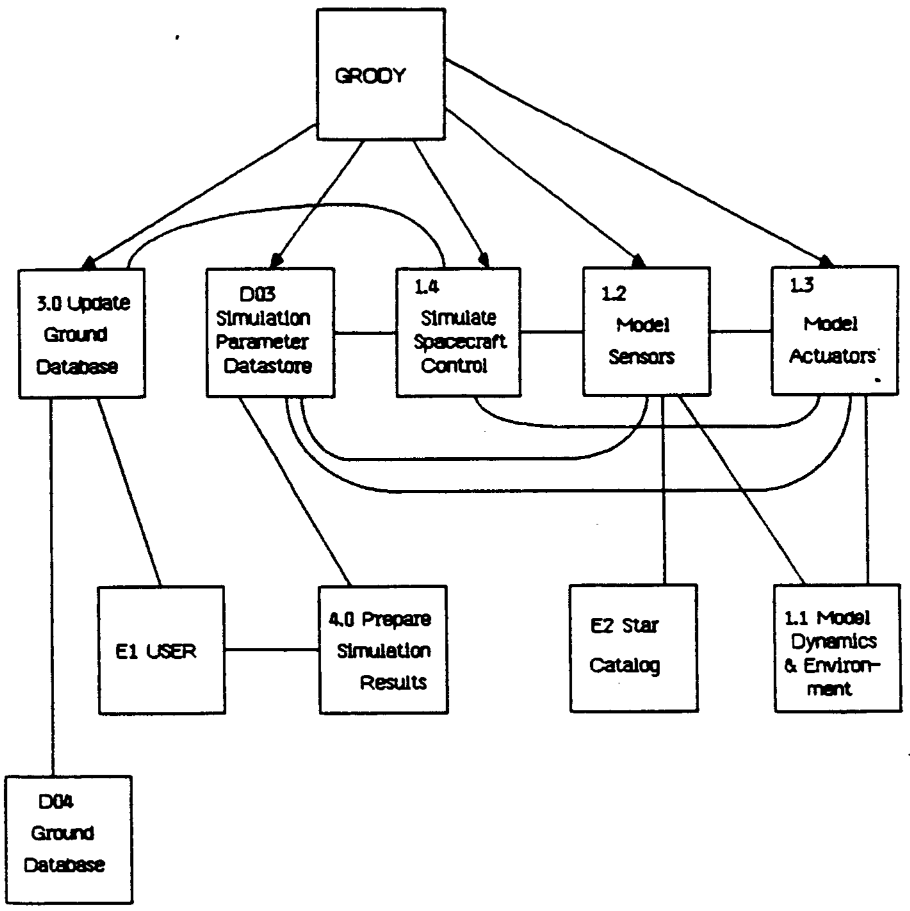
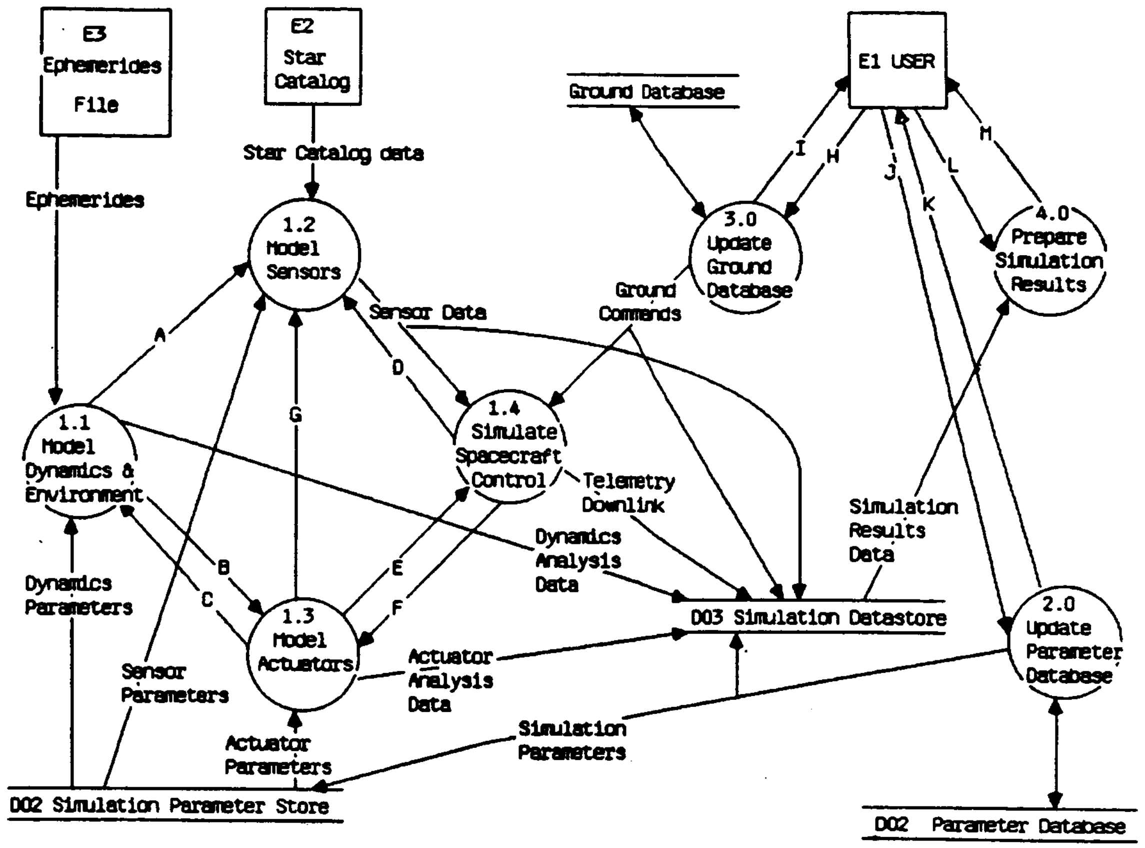

GRODY
3.0 Update Ground Database
D03 Simulation Parameter Datastore
1.4 Simulate Spacecraft Control
1.2 Model Sensors
1.3 Model Actuators
E1 USER
4.0 Prepare Simulation Results
E2 Star Catalog
1.1 Model Dynamics & Environment
D04 Ground Database

Figure 5-6. Next Level of Entities

This process continues until the ends of the afferent and efferent data flows are reached. Figure 5-7 is a recast data flow diagram for GRODY. This diagram is a releveling of the original data flow diagrams to reflect support of the central entity. As before, we have used the shorthand labels for the data flows. This diagram will later be used in identifying objects.



Dataflow Labels

A:	Sensor Reference Data • Attitude • Angular Velocity
B:	Center of Mass • Geomagnetic Field in BCS
C:	Wheel Angular Momenta • Array & Antenna Angles
D:	Sensor Commands
E:	Hardware Status
F:	Actuator Commands
G:	Wheel Speeds + Torquer Dipole
H:	Ground Database Options
I:	Ground Database Report
J:	Parameter Database Options
K:	Parameter Database Report
L:	Results Processing Options
M:	Simulation Results

Figure 5-7. Recast GRODY Data Flow Diagram

Figure 5-8 is an initial entity graph for GRODY. In Figure 5-8 we show the RUN operation flowing from the user to the most senior entity GRODY. This signal would come from outside the entity graph if GRODY were started by the operator or by the system when it is powered up. The RUN signal and the control flowing from GRODY are the only edges on the entity graph that now have direction. Assignment of direction to the other edges is discussed in the next subsection.

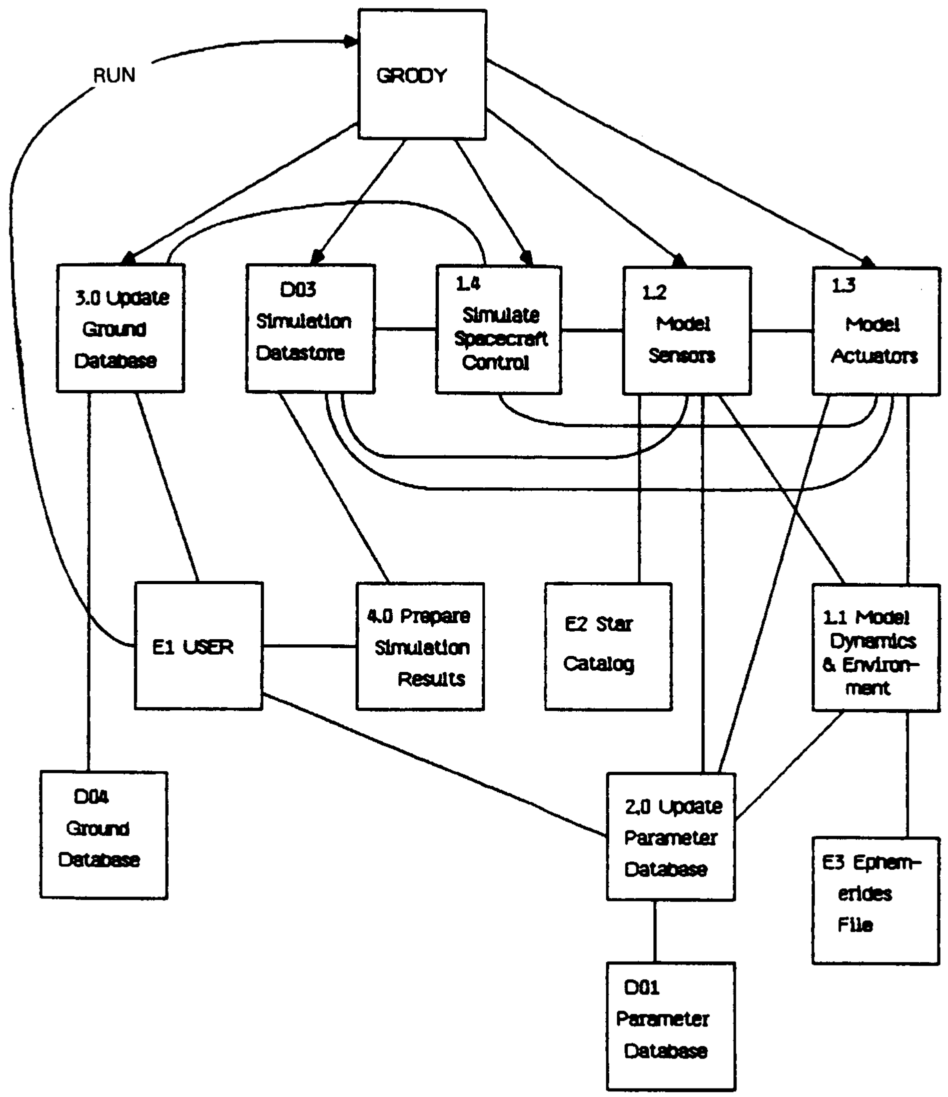


Figure 5-8. GRODY Entity Graph

What we have shown so far is for illustration. An actual design would be derived in fewer steps. A complete recast data flow diagram could be drawn after the central entity is identified, and then the initial entity graph can be drawn using the "inside out" method described above. In the example shown this far, the initial entity graph will be extensively modified before the final objects are found. When a recast data flow diagram is drawn before identifying entities the relationships between processes and states are easier to see. This should make the entities identified closely related to the final objects. In some cases it is possible to identify objects directly from a recast data flow diagram.

5.2 IDENTIFYING OBJECTS

The first step in identifying objects from an entity graph is to add directions of control where the problem determines the control flow. Figure 5-9 is the GRODY entity graph with these modifications. The database entities and external entities are "passive," so they all have control flowing into them. The idea of the USER being "controlled" runs against the intuitive idea of a user controlling software, but in the sense of control flow what happens is that a software system will call an operation such as TEXT IO.GET to find out what the user wants to do.

SIMULATE SPACECRAFT CONTROL is required to give simulated control commands. This implies that MODEL SENSORS and MODEL ACTUATORS are junior to SIMULATE SPACECRAFT CONTROL. UPDATE PARAMETER DATABASE is made senior to 1.1, 1.2, and 1.3 so that the user can control the state of these entities. We have not added the corresponding control flow between UPDATE GROUND DATABASE and 1.4 SPACECRAFT CONTROL because we have not determined whether ground commands will be requested by 1.4 or whether they will be provided from outside. No direction of control is identified between MODEL SENSORS, MODEL ACTUATORS, and MODEL DYNAMICS & ENVIRONMENT. Nothing in the problem domain determines direction of control among these three entities. We will be able to choose these directions of control later based on virtual machine hierarchy considerations.

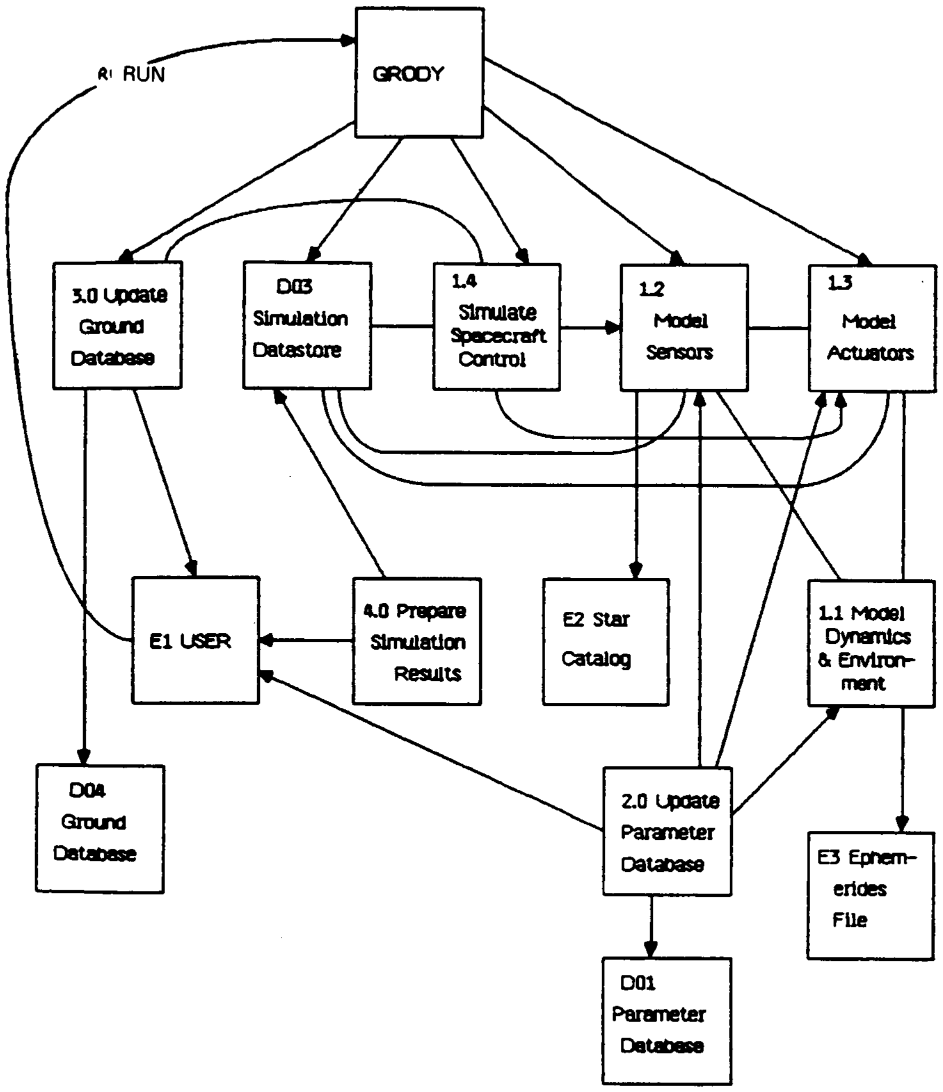


Figure 5-9. Entity Graph With Control Flows

The next step is to identify objects and to place them in a strong seniority hierarchy. We want to balance the level of abstraction of each object, the desire for a good seniority hierarchy, and the complexity of relationships between objects.

In our GRODY example, we can see from Figure 5-9 that 2.0, 3.0, and 4.0 are all senior to the user, and that USER, GRODY, and 3.0 form a cyclic graph. This means that these five entities are all on the same virtual machine level. Entities 2.0, 3.0, and 4.0 all control databases, with the first two having sole control of the PARAMETER DATABASE and GROUND DATABASE, respectively. Since they all interact with the user, we create a USER INTERFACE by combining 2.0, 3.0, 4.0, D01, and D04. Combining the user and database interactions into a single object provides good entity abstraction. We will see later that D03 is not contained in USER INTERFACE due to virtual machine hierarchy considerations. The processes and datastores in USER INTERFACE are circled on the recast data flow diagram (see Figure 5-10).

We chose the process 1.4 SIMULATE SPACECRAFT CONTROL as the central entity because it contains the control laws being tested by the simulator. This same consideration dictates the use of a separate SPACECRAFT CONTROL object. Process bubble 1.4 on the recast data flow diagram is circled to reflect this decision (see Figure 5-10 again). We still have not chosen whether USER INTERFACE or SPACECRAFT CONTROL will be a senior object, nor do we want to until all the objects are identified.

Entities 1.1, 1.2 and 1.3 pose a slightly more difficult problem. One alternative is to combine 1.2 MODEL SENSORS and 1.3 MODEL ACTUATORS into an ATTITUDE HARDWARE object. We can then make 1.1 a junior object so that 1.4 controls ATTITUDE HARDWARE which in turn controls 1.1 MODEL DYNAMICS & ENVIRONMENT. This hierarchy is one way of producing layers of virtual machines. Another alternative is to combine 1.1,

ORIGINAL PAGE IS
OF POOR QUALITY

1.2 and 1.3 into a single object. We will call this object TRUTH MODEL because it provides "true" responses to control commands. In this case deciding the flow of control between entities 1.1, 1.2 and 1.3 is defered until the child object diagram for TRUTH MODEL is generated. The first alternative yields objects with higher abstraction, but the second will give a simpler design. The TRUTH MODEL object has abstraction somewhere between entity (model true spacecraft response) and action (model the related actions of sensors, actuators, dynamics and environment) abstraction. Thus we choose the second alternative as "abstract enough" and as part of a good virtual machine hierarchy. Again, the processes and datastores contained by the object are circled on the recast data flow diagram (Figure 5-10).

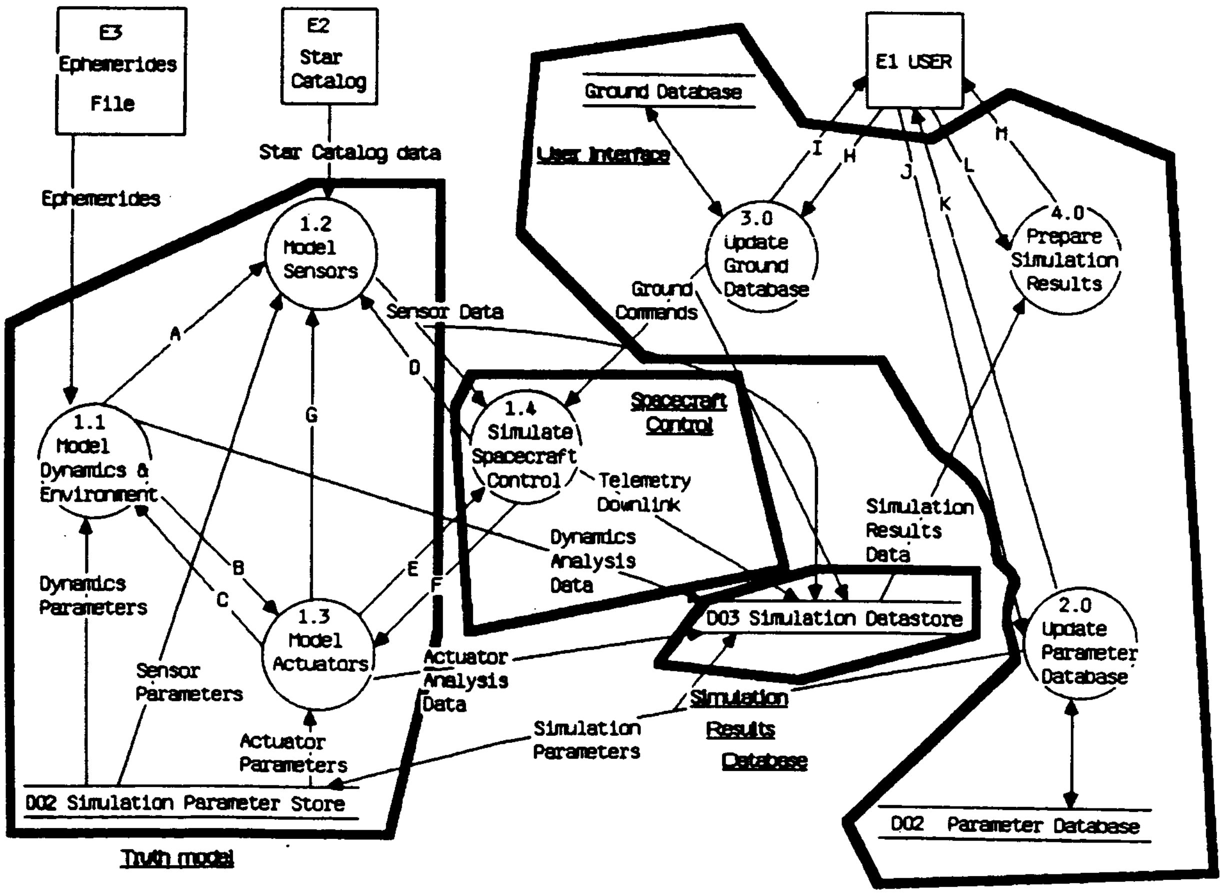


Figure 5-10. Recast GRODY DFD With Object Boundaries Shown

The data store D03 SIMULATION DATASTORE has not yet been associated with an object. It could be placed within the USER INTERFACE, but that would result in USER INTERFACE both calling TRUTH MODEL to initialize simulation parameters and being called by TRUTH MODEL to store results data. This is not necessarily bad, but we would like to avoid this situation if it is possible to do so. To preserve our virtual machine hierarchy we define a SIMULATION RESULTS DATABASE that is junior to everybody. We have lost some of the abstraction by splitting this object from USER INTERFACE, but both objects are still good abstractions, and we have gained a better control hierarchy. Figure 5-10 is now completed by circling the D03 SIMULATION DATASTORE.

Figure 5-11 is the object diagram resulting from the above analysis. We have chosen to make the USER INTERFACE senior to SPACECRAFT CONTROL, but the arrow between these two objects could be reversed and we would still have a virtual machine hierarchy. The decision to make USER INTERFACE the senior object was based on the need to have the user control a simulation. The USER INTERFACE object "controls" the user by calling a read operation to get data or user options, and then calls on the other simulator components to perform the operation requested. The important concept here is that the decision was not made on the basis of the design rules discussed above, but rather on what would be a more desirable way to meet the specification.

Figure 5-11 shows a clear seniority hierarchy, but decisions still need to be made about how strong this hierarchy will be. Any changes can be made to Figure 5-11 that leave paths available for data to flow from a source to its correct destination. We can eliminate the communications among USER INTERFACE, SIMULATE SPACECRAFT CONTROL and TRUTH MODEL to get the design shown in Figure 5-12, which is loosely coupled and highly structured at its senior level. Alternatively,

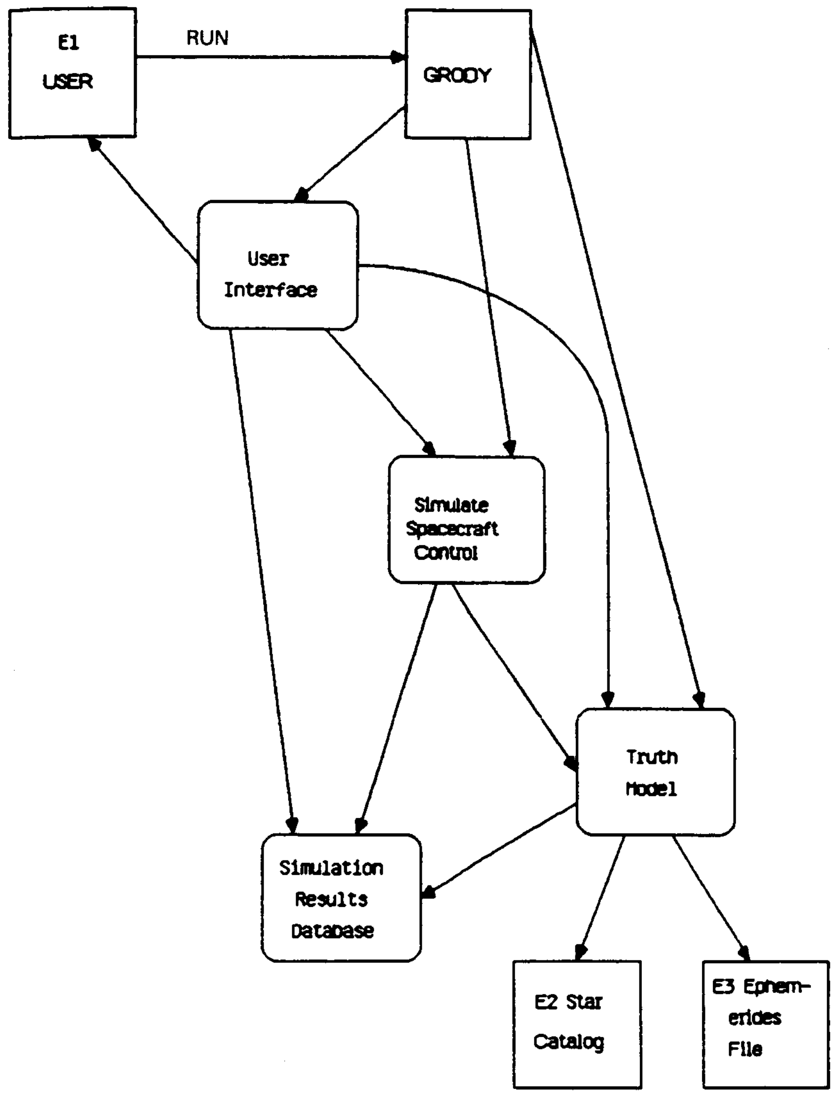


Figure 5-11. Initial Object Diagram

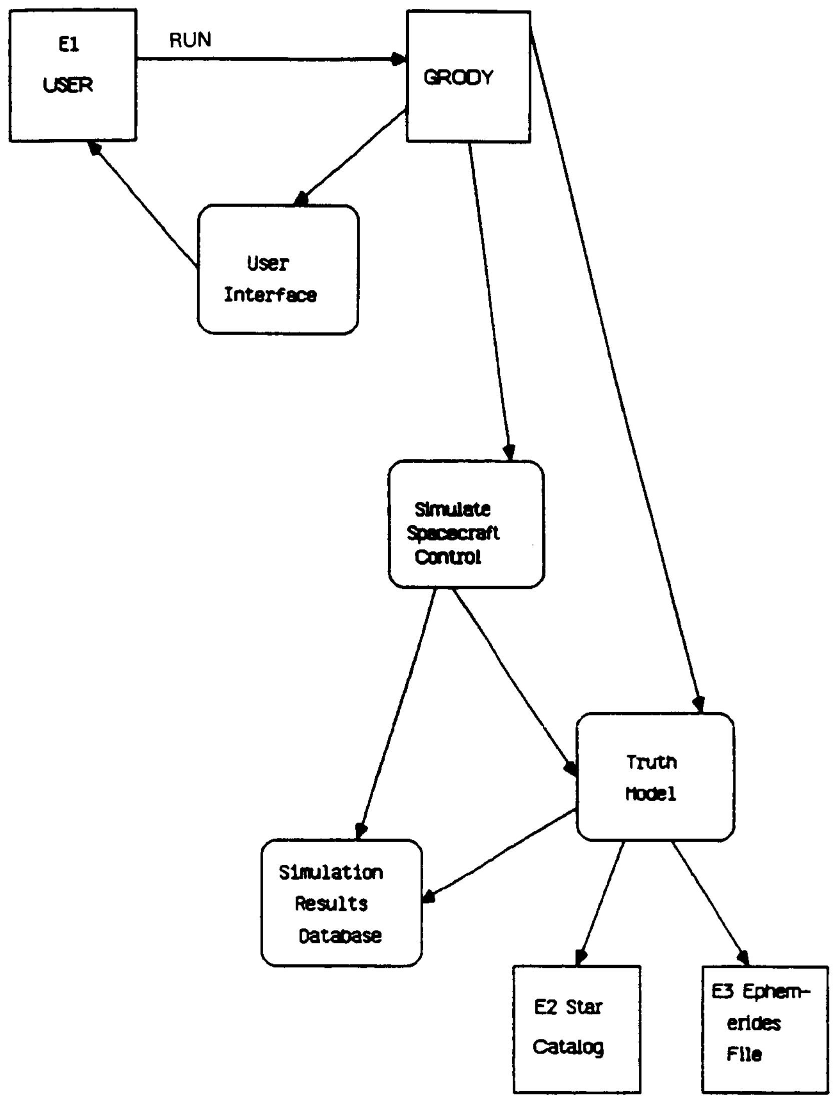


Figure 5-12. More Centralized GRODY Design

we can combine GRODY and USER INTERFACE to give a design with more decentralized control, as in Figure 5-13. A third choice is to keep GRODY as an entity that performs scheduling but that does not exchange data with junior virtual machine levels. Then Figure 5-11 would stand as the final object diagram. We choose the decentralized configuration of Figure 5-13 because we want to eliminate the bottlenecks that can be caused by a complex central control entity. The seniority hierarchy is still strong, but all data "fly non-stop" from their source to the destination.

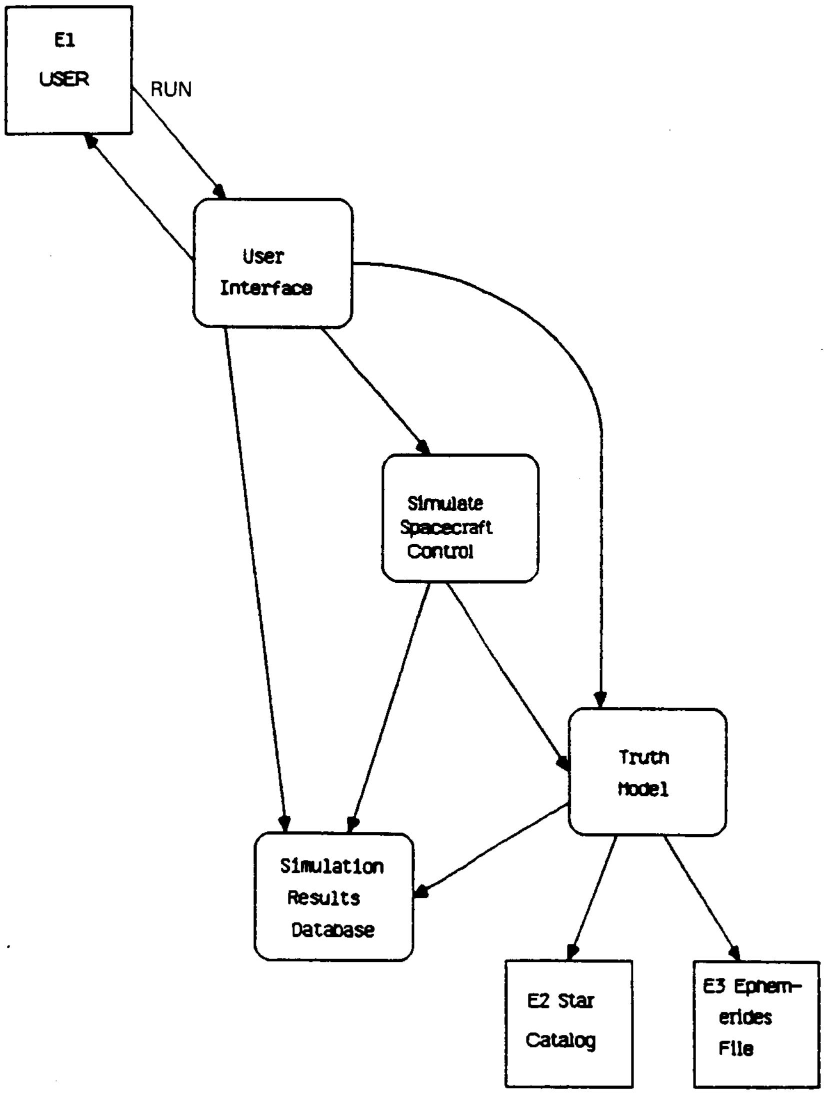


Figure 5-13. Less Centralized GRODY Design

The entities on the initial object diagram are now either objects or external entities, and we have fully considered flow of control issues. Before we can go on and identify operations and complete the object diagram we must consider objects that are required but not visible from the analysis of a data flow diagram. For GRODY the only major requirement we have not handled is scheduling and keeping track of simulated time. Two alternatives are to design from Figure 5-13 and to have the "most senior" object GRODY handle the scheduling and the timing; or to create a timer object junior to SPACECRAFT CONTROL which will update the simulated time. In the second case the scheduling is implicit in the response of junior objects to requests and commands from SPACECRAFT CONTROL. Figure 5-14 is the object diagram generated by using this option. We choose the second option as a more decentralized design.

5.3 DESIGN USING ABSTRACTION ANALYSIS

Considering required objects completes the process of object identification. The next step is to formally map the specifications to objects and then to identify operations. Experienced designers can actually shorten the object identification process from what is shown above by skipping the explicit use of entity graphs. The steps that are then taken are as follows:

1. Identify central entity.
2. Draw a recast data flow diagram.
3. Identify objects and draw boundaries on recast DFD.
4. Draw an object diagram with a hierarchy that best balances requirements for loosely coupled objects and for the elimination of data and control bottlenecks.

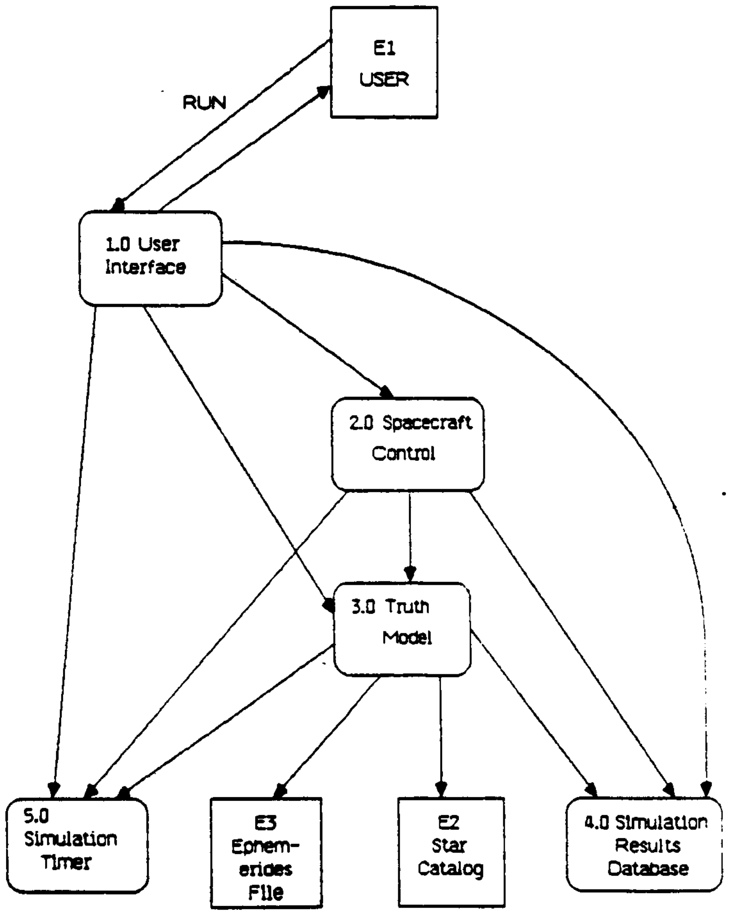


Figure 5-14. GRODY Object Diagram

If entity graphs are used they are drawn as an intermediate stage between steps 2 and 3.

After drawing the object diagram the design process consists of the following:

1. Generate an object contents table.
2. Identify operations.
3. Label concurrent objects by adding simultaneous control.
4. Generate data flow diagram for each object on the object diagram.

To make this a recursive process, the part of the recast data flow diagram describing each object is used, along with the associated lower level data flow diagrams, as a starting point for the identification of child objects. The object's operations provide central entities which the designer uses as a starting point for drawing an entity graph for the object. This graph is then used to identify child objects, and then the four steps above are taken to complete a child object diagram. Later we will construct the child object diagram for the TRUTH MODEL object on the GRODY level 0 object diagram (Figure 5-14, object 3.0). The steps listed above are described in more detail in the following subsections.

5.3.1 GENERATING OBJECT CONTENTS TABLES

The object contents table lists the objects on a diagram, the processes that each object will implement, the states hidden by each object on the diagram, and system considerations that are not captured by the data flow diagrams. Figure 5-15 shows the object contents table for the GRODY level 0 object diagram. The processes are listed to the level of detail needed to show the boundaries between objects. The states and system considerations are also shown at an appropriate level. In Figure 5-14 the TRUTH MODEL object contains processes 1.1, 1.2, and 1.3 and SIMULATE SPACECRAFT CONTROL contains 1.4. Thus we have to break up process 1.0 when we write the object contents table. In Figure 5-15 the states hidden are all data stores, but they can also be data elements or data records that are a subset of a data store. It is necessary to show a hidden state on the object contents table only when it is visible on a data flow diagram showing the interior of an object. For example, the object SPACECRAFT CONTROL certainly has internal state, but this state is not visible at this level.

User Interface
- Processes:
  - 2.0 Update Parameter Database
  - 3.0 Update Ground Database
  - 4.0 Prepare Simulation Results
- States:
  - D01 Parameter Database
  - D04 Ground Database

Spacecraft Control
- Processes:
  - 1.4 Spacecraft Control
- System Considerations
  - required as object to test control laws

Truth Model
- Processes
  - 1.1 Model Dynamics & Environment
  - 1.2 Model Sensors
  - 1.3 Model Actuators
- States
  - D02 Simulation Parameter Store

Simulation Results Database
- State
  - D03 Simulation Datastore

Timer
- System Considerations
  - Simulator is required to keep a simulated time

Figure 5-15. Object Contents Table for GRODY

5.3.2 USING THE RECAST DATA FLOW DIAGRAM WITH OBJECT BOUNDARIES

Figure 5-10 shows the diagram for GRODY with the object boundaries drawn on top. In most cases object boundaries can be drawn around processes and data stores on the recast data flow diagram. If this is not possible, the child data flow diagrams or the data dictionary must be examined, and the parent process or data store must be divided among the appropriate objects. This kind of adjustment will be demonstrated in more detail as we break down the TRUTH MODEL.

5.3.3 IDENTIFYING OPERATIONS

Identifying operations is a continuation of the direction-of-control analysis done for the entity graph. We use the direction of control that has been established, the data flows across object boundaries, and the processes and states (data stores) connected by these data flows. Child data flow diagrams and data dictionary entries are used to gain more details on the processes and data involved.

For example, Figure 5-10 shows the data exchanged between SPACECRAFT CONTROL and TRUTH MODEL. These data are generated by operations modeling sensor and actuator behavior. These operations are provided by TRUTH MODEL and used by SPACECRAFT CONTROL. Figure 5-10 shows that 1.2 MODEL SENSORS and 1.4 SIMULATE SPACECRAFT CONTROL communicate using the data flows "Sensor Data" and "Sensor Commands". Examining the child data flow diagram for 1.2 reveals the exact sensors used and the related data and commands. Examining data flow diagram 1.2 (Figure 5-16) shows the details of what sensor processes exist. The operations break down into categories of getting sensor data and (in the case of gyros and FHST) processing sensor commands. Similarly the interface between 1.3 MODEL ACTUATORS and 1.4 SIMULATE SPACECRAFT CONTROL can be characterized by operations that command actuators. Using these data flow interfaces we can begin to construct operation definitions for the TRUTH MODEL operations used by SPACECRAFT CONTROL. The operations provided by the other objects are derived in the same way as the TRUTH MODEL operations. These can then be combined into complete object descriptions. Figure 5-17 shows the complete object description for TRUTH MODEL. Note that we have also described the purpose of the TRUTH MODEL in Figure 5-17 to further document the object description.

Adjustments can be made to objects and operations even at this late stage. For example we see on Figure 5-10 that the data flows "Ground Commands" and "Simulation Parameters" enter the SIMULATION RESULTS DATABASE from the USER INTERFACE. These data are also returned to the USER INTERFACE as part of "Simulation Results Data." We can move

** 1.2 MODEL SENSORS **

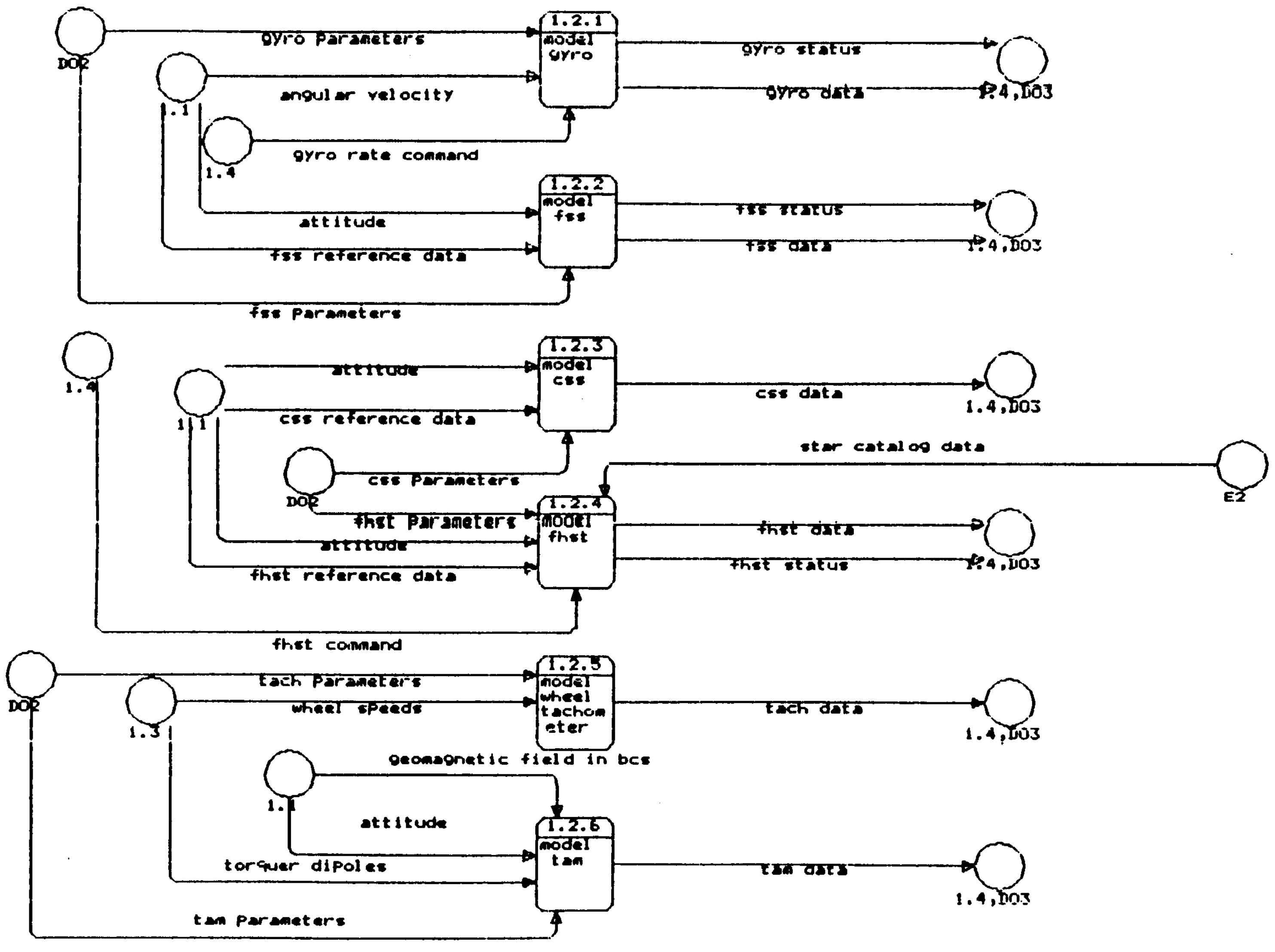


Figure 5-16. 1.2 Model Sensors

TRUTH-MODEL

Purpose:

This object simulates the "true response" of the spacecraft to attitude control commands. It processes actuator commands, generates simulated sensor output and integrates the spacecraft attitude dynamics equations. It includes models of environmental perturbations and sensor measurement noise.

Provides:

```
RESET ()
INITIALIZE-PARAMETERS (TRUTH-MODEL-PARAMETERS)

GET-GYRO-DATA () GYRO-STATUS + GYRO-DATA
GET-FSS-DATA () FSS-STATUS + FSS-DATA
GET-CSS-DATA () CSS-DATA
GET-FHST-DATA () FHST-STATUS + FHST-DATA
GET-TACH-DATA () TACH-DATA
GET-TAM-DATA () TAM-DATA

COMMAND-GYRO (GYRO-RATE-COMMAND)
COMMAND-FHST (FHST-COMMAND)

COMMAND-THRUSTERS (THRUSTER-COMMAND)
COMMAND-REACTION-WHEELS (WHEEL-COMMAND)
COMMAND-TORQUERS (TORQUER-COMMAND)
```

Uses:

```
E2 STAR-CATALOG
  GET-STAR-DATA

E3 EPHEMERIDES-FILE
  GET-EPHEMERIDES

4.0 SIMULATION-PARAMETERS-DATABASE
  PUT-RESULTS-DATA

5.0 SIMULATION-TIMER
  GET-TIME
```

Figure 5-17. Truth Model Object Description

the appropriate parts of the data store D03 SIMULATION DATASTORE into the USER INTERFACE by updating the object contents table. The states OUTPUT SIMULATION PARAMETERS and OUTPUT GROUND COMMANDS are added to USER INTERFACE, and the contents of SIMULATION RESULTS DATABASE are updated to reflect the removal of these data. The data dictionary is used to maintain consistency in defining states contained by an object, in the same way that the different levels of data flow diagram are used to define what processes are contained by an object. The actual change to data flow diagrams can be made when we start decomposing our objects. Figure 5-18 shows the updated object contents table.

User Interface

- **Processes:**
  - 2.0 Update Parameter Database
  - 3.0 Update Ground Database
  - 4.0 Prepare Simulation Results
- **States:**
  - D01 Parameter Database
  - D04 Ground Database
  - Output Simulation parameters • Output Ground Commands

Spacecraft Control

- **Processes:**
  - 1.4 Spacecraft Control
- **System Considerations**
  - required as object to test control laws

Truth Model

- **Processes**
  - 1.1 Model Dynamics & Environment
  - 1.2 Model Sensors
  - 1.3 Model Actuators
- **States**
  - D02 Simulation Parameter Store

Simulation Results Database

- **State**
  - Sensor Data • Telemetry Downlink •
  - Dynamics Analysis Data • Actuator Analysis Data

Timer

- **System Considerations**
  - Simulator is required to keep a simulated time

Figure 5-18. Updated Object Contents Table for GRODY

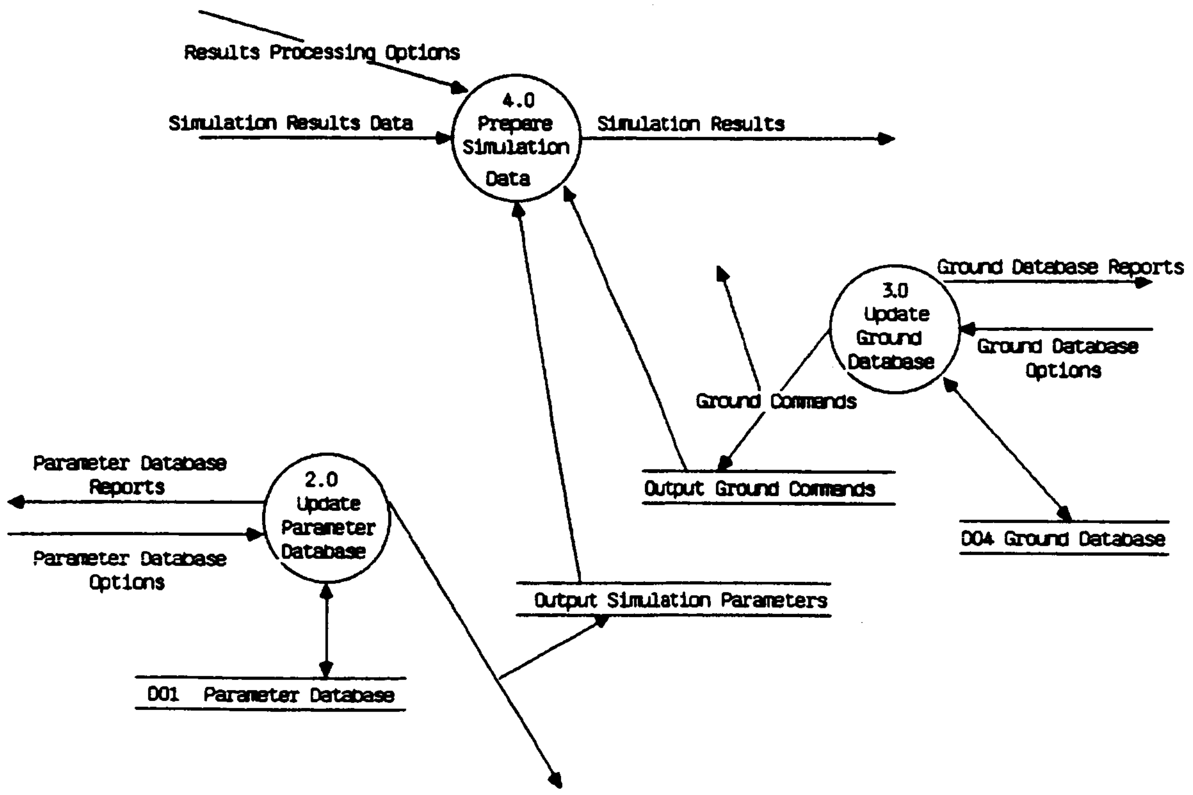
The last step in generating an object diagram is to produce separate data flow diagrams for each object. These are used when the child object diagrams are created. Figures 5-19 and 5-20 show these diagrams for USER INTERFACE and TRUTH MODEL, respectively. Note that these diagrams are not merely the segments circled in Figure 5-10, but that they reflect the up-to-date object contents, as shown in Figure 5-17. The object SPACECRAFT CONTROL contains only process 1.4 SIMULATE SPACECRAFT CONTROL, which leaves the child data flow diagram for process 1.4 as the starting point for object identification. SIMULATION RESULTS DATABASE is a state that has no child object diagram, thus no data flow diagram is necessary for this object. If an object encapsulates a state, the data dictionary will give the details needed to complete the design. If the data store on a data flow diagram represents a more sophisticated data structure (such as a queue) a child object diagram will have to be generated to show how the data structure is to be implemented and what operations can be performed.

The TIMER object was generated from a non-functional requirement. As in the example of a data store representing a queue, we have to use the operations identified to generate a child object diagram. TIMER is simple enough to generate a child object diagram directly from the knowledge of what the operations are supposed to do. For a more complex object we would consider generating a specification for that object using data flow diagrams before attempting a more detailed design.

5.3.4 GENERATING CHILD OBJECT DIAGRAMS

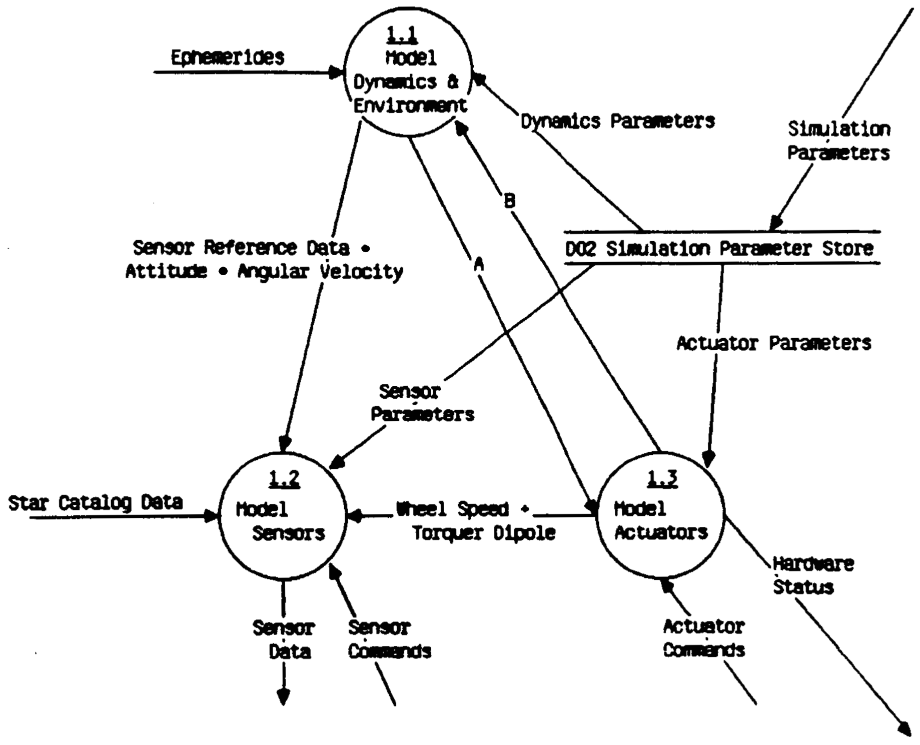
The production of the TRUTH MODEL object diagram will show how an object's operations provide a starting point for the next level of design. Figure 5-20 shows the processes and

ORIGINAL PAGE IS
OF POOR QUALITY



Results Processing Options
Simulation Results Data
4.0 Prepare Simulation Data
Simulation Results
Ground Database Reports
3.0 Update Ground Database
Ground Database Options
Ground Commands
Output Ground Commands
D04 Ground Database
Parameter Database Reports
Parameter Database Options
2.0 Update Parameter Database
Output Simulation Parameters
D01 Parameter Database

Figure 5-19. User Interface Data Flow Diagram



A: Center of Mass • Geomagnetic field in BCS
B: Wheel Angular Momentum • Control Torque • Array and Antenna Angles

Figure 5-20. Truth Model Data Flow Diagram

states contained by TRUTH MODEL. Since the sensors and actuators (1.2 and 1.3) directly support the operations provided to SPACECRAFT CONTROL by TRUTH MODEL the first step is to make these into entities and to make them the most senior entities within TRUTH MODEL. The process 1.1 MODEL DYNAMICS AND ENVIRONMENT clearly contains at least two entities, one to model the dynamics and one to model the environment. Thus we examine the child data flow diagram (Figure 5-21) to see what the next level provides in the way of entities. Process 1.1.2 COMPUTE ENVIRONMENTAL TORQUES is a process which models the effect of the spacecraft environment on attitude dynamics, and 1.1.4 MODEL INTERNAL MOTION models the effect of moving spacecraft parts on the attitude dynamics. Thus combining 1.1.2, 1.1.3, and 1.1.4 into a single ATTITUDE DYNAMICS entity is a reasonable abstraction. 1.1.1 COMPUTE EPHEM DEPENDENT PARAMETERS then becomes the SPACECRAFT ENVIRONMENT entity.

To finish the entity graph we only need to decide whether the data store SIMULATION PARAMETER STORE should be divided among the already identified entities, or whether it should become an entity itself. We choose the latter and draw an entity graph (Figure 5-22). This choice is opposite to what we did for D02 SIMULATION PARAMETER STORE when we generated the level 0 entity graph. Designers will make such changes in their approach as more details of the problem become apparent.

In Figure 5-22, control flow is shown for the external entities and the SIMULATION PARAMETER STORE entity. These entities must show control flowing towards them since they contain data but no processing. In addition, the parent object diagram shows that SENSORS and ACTUATORS provide data as they are requested by the SPACECRAFT CONTROL. Thus, the SENSORS and ACTUATORS in turn need to request data to complete the actions required. Again, this is determined not

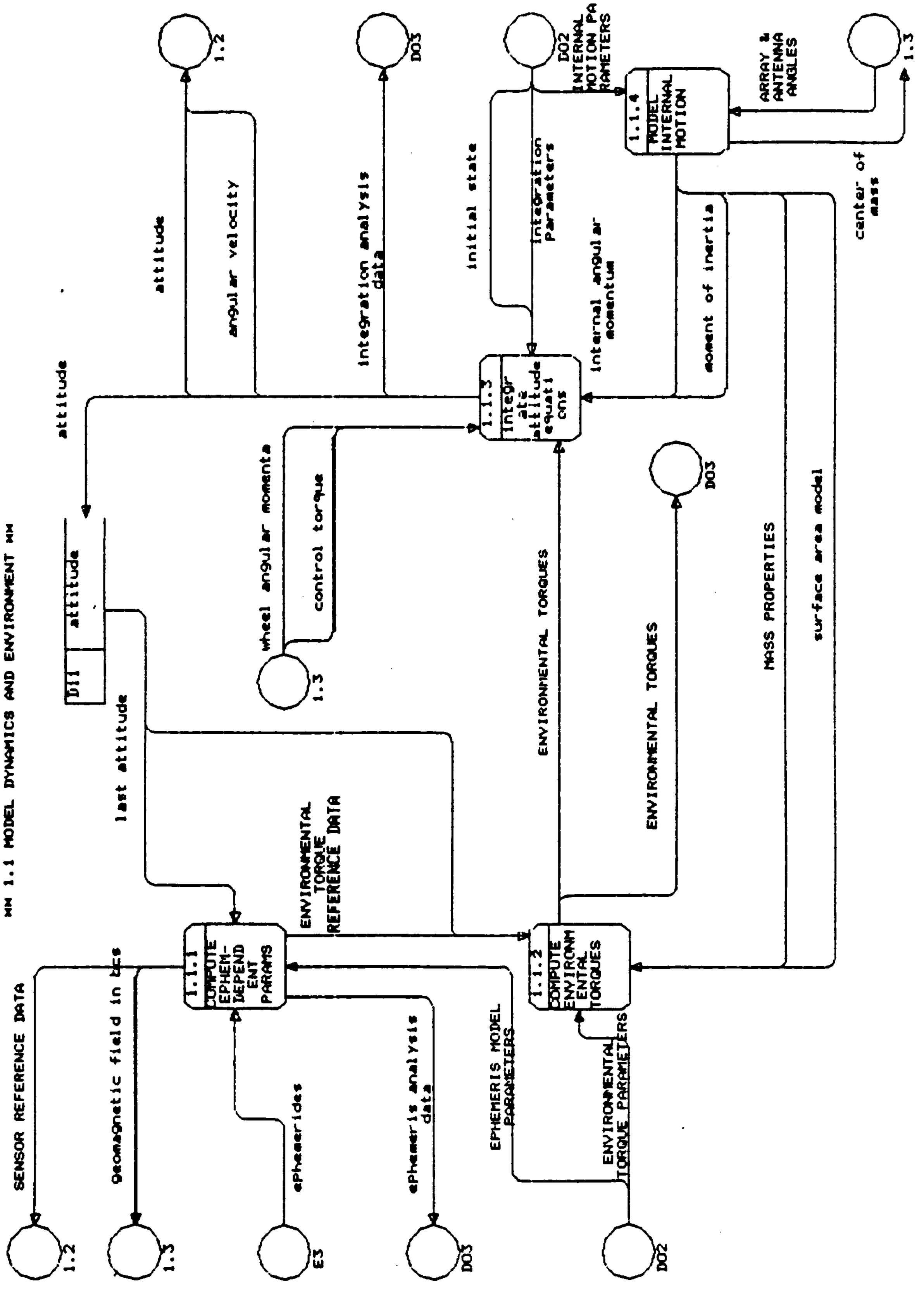

** 1.1 MODEL DYNAMICS AND ENVIRONMENT **
1.1.1 COMPUTE EPHEM-DEPENDENT PARAMS
1.1.2 COMPUTE ENVIRONMENTAL TORQUES
1.1.3 Integrate attitude equations
1.1.4 MODEL INTERNAL MOTION
D11 attitude
SENSOR REFERENCE DATA
Geomagnetic field in bcs
ephemerides
ephemeris analysis data
EPHEMERIS MODEL PARAMETERS
ENVIRONMENTAL TORQUE PARAMETERS
ENVIRONMENTAL TORQUE REFERENCE DATA
last attitude
ENVIRONMENTAL TORQUES
MASS PROPERTIES
surface area model
wheel angular momenta
control torque
attitude
angular velocity
integration analysis data
initial state
integration parameters
internal angular momentum
moment of inertia
center of mass
ARRAY & ANTENNA ANGLES
D02 INTERNAL MOTION PARAMETERS
1.2
1.3
E3
D03
D02

Figure 5-21. 1.1 Model Dynamics and Environment by the topology of the entity graph but by an overall design strategy. This leaves only the direction of control between SENSORS and ACTUATORS and between ATTITUDE DYNAMICS and SPACECRAFT ENVIRONMENT to be determined. In the latter case we make the SPACECRAFT ENVIRONMENT a junior object. This is because the dynamics modeling is likely to change from mission to mission, while the environment does not change. This will allow SPACECRAFT ENVIRONMENT to be implemented as a library unit and to be reused for subsequent missions. The SENSORS and ACTUATORS are combined into a single ATTITUDE HARDWARE object. This is a good abstraction, and it allows us to simplify the TRUTH MODEL design and to defer consideration of the sensor/actuator interactions to the next level of detail. Figure 5-23 is the TRUTH MODEL object diagram.

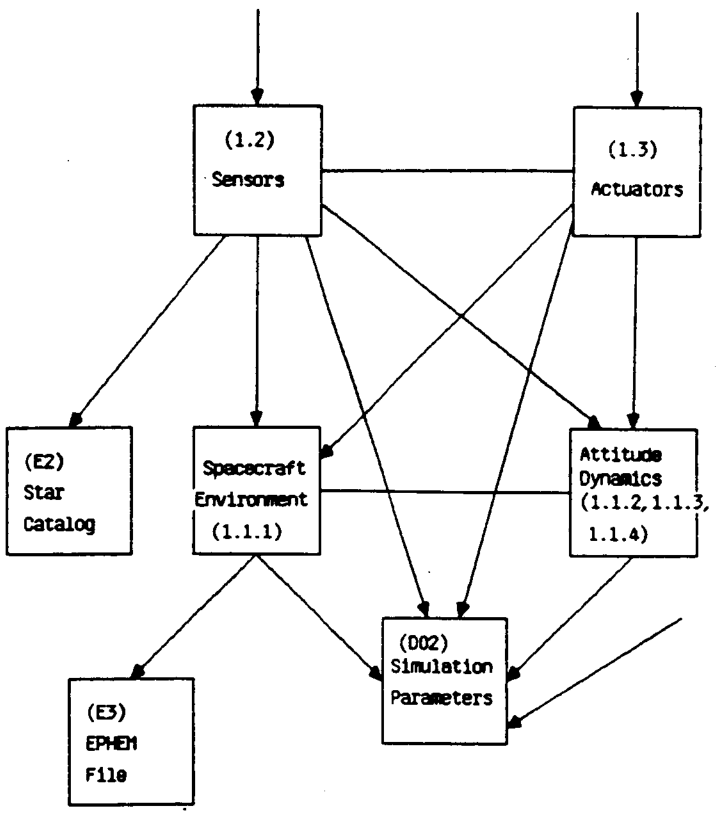


Figure 5-22. Truth Model Entity Graph

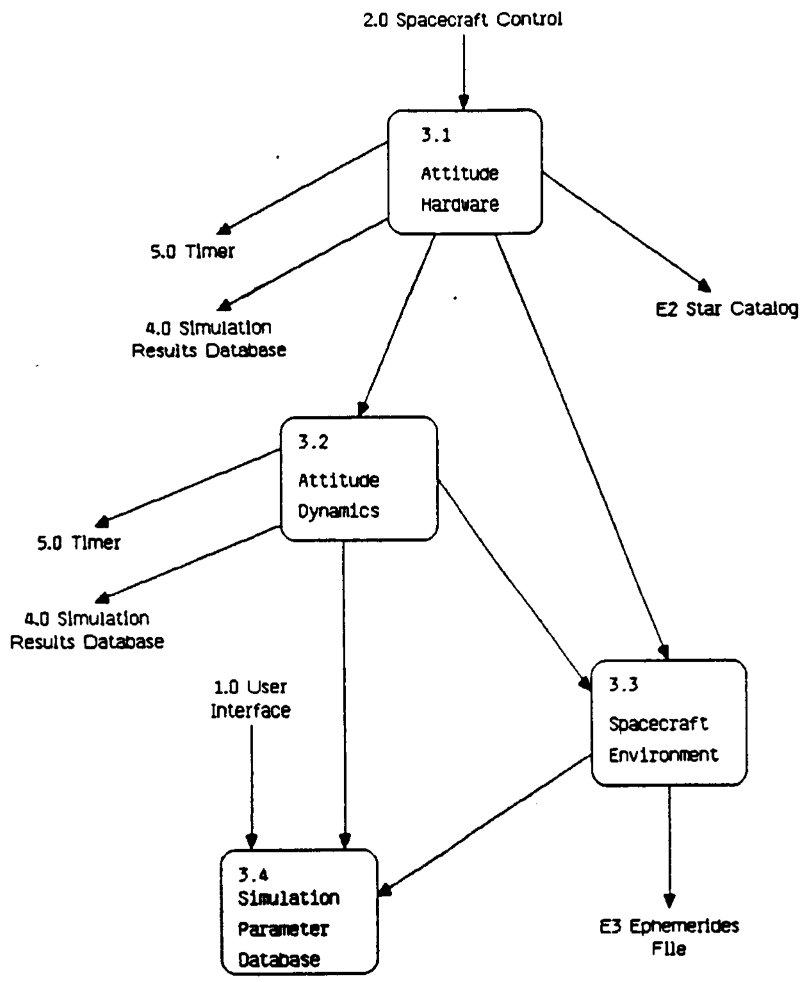


Figure 5-23. Truth Model Object Diagram

The object contents table and operations dictionary entries are generated in exactly the same way as for the level 0 object diagram. The only difference is in how to start identifying the objects from a data flow diagram.

5.3.5 TASKING CONSIDERATIONS

In Section 3 we showed concurrency on an object diagram by having a single operation (e.g., RUN) flow into two or more objects. The same notation can be used on entity graphs. We thus have a means of representing concurrency on entity graphs, but at this point we have not yet developed guidelines for concurrency within our methodology. Cherry's [Cherry 85a] criteria for determining when an entity is concurrent can also be used within the abstraction analysis model. In short, we have not imposed any rigorous guidelines about determining when objects are concurrent, but have a notation that is flexible enough to represent concurrency throughout the transition between specification and design.

SECTION 6 - CONCLUSION

Object diagrams, abstraction analysis and associated principles provide a unified framework which encompasses concepts from several other methodologies [Yourdon 79, Booch 83, Cherry 85b]. The use of object diagrams and abstraction analysis provides the following:

- A general object-oriented approach which handles system design from the top level, through object-oriented decomposition, down to a completely functional level.
- A method of tracing how a design meets the specification.
- A design notation that maps into Ada, thus providing a composite mapping from a specification to Ada software.
- A design notation flexible enough to represent both traditional structured designs and non-hierarchical designs such as those produced using PAMELA.
- Criteria for partitioning a software system into modules and for choosing direction of control.
- Support for walkthroughs and iterative refinement of a design through the use of graphical notation for both the specification and the design.

The concepts discussed in this report form an integral part of an object-oriented software development life cycle. We are currently studying how object-oriented concepts can be used in other phases of the life cycle, such as specification and testing. When complete, this synthesis should produce a truly general object-oriented development methodology.

REFERENCES

[Agresti 86] W. W. Agresti, V. E. Church, D. N. Card and P. L. Lo. "Designing with Ada for Satellite Simulation: A Case Study", Proc. of the 1st Intl. Conf. on Ada Applications for the Space Station, June 1986.

[Booch 83] Grady Booch. Software Engineering with Ada, Menlo Park: Benjamin/Cummings, 1983.

[Cherry 85a] George W. Cherry. PAMELA: Process Abstraction Method for Embedded Large Applications, Course notes, Thought**Tools, January 1985.

[Cherry 85b] George W. Cherry and Bard S. Crawford. The PAMELA (tm) Methodology, November 1985.

[DeMarco 78] Tom DeMarco. Structured Analysis and System Specification, Prentice-Hall, New York, 1978.

[Dijkstra 68] Edsgar W. Dijkstra. "The Structure of the 'THE' Multiprogramming System," Comm. of the ACM, May 1968.

[Goldberg 83] Adele Goldberg and David Robson. Smalltalk 80: the Language and Its Implementation, Addison-Wesley, 1983.

[Ledgard 77] Henry F. Ledgard and Robert W. Taylor. "Two Views of Data Abstraction," Comm. of the ACM, June 1977.

[Liskov 74] Barbara H. Liskov and S. N. Zilles. "Programming with Abstract Data Types," Proc. of the ACM Symp. on Very High Level Languages, SIGPLAN Notices, April 1974.

[Nelson 86] Robert W. Nelson. "NASA Ada Experiment--Attitude Dynamic Simulator," Proc. of the Washington Ada Symposium, March 1986.

[Parnas 72] David L. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules," Comm. of the ACM, December 1972.

[Rajlich 85] Vaclav Rajlich. "Paradigms for Design and Implementation in Ada," Comm. of the ACM, July 1985.

[Seidewitz 85a] Ed Seidewitz. Object Diagrams, unpublished GSFC report, May 1985.

[Seidewitz 85b] Ed Seidewitz. Some Principles of Object-Oriented Design, unpublished GSFC report, August 1985.

[Seidewitz 86] Ed Seidewitz and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," Proc. of the 1st Intl. Conf. on Ada Applications for the Space Station, June 1986.

[Stark 86] Mike Stark. Abstraction Analysis: From Structured Specification to Object-Oriented Design, unpublished GSFC report, April 1986.

[Yourdon 79] Edward Yourdon and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, 1979.

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu and D. S. Wilson, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, May 1985

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-306, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1985

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-83-104, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) User's Guide, T. A. Babst, W. J. Decker, P. Lo, and W. Miller, August 1984

SEL-83-105, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) System Description, P. Lo, W. J. Decker, and W. Miller, August 1984

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

Agresti, W. W., Tutorial: New Paradigms for Software Development. New York: IEEE Computer Society Press, July 1986

Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[3]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1979

[3]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[1]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[3]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[3]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

[1]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

[1]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering, August 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland Technical Report, TR-1501, May 1985

Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

[2]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

[1]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[3]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[3]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[3]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: IEEE Computer Society Press, 1978

[1]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

[1]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering, August 1985

[3]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: IEEE Computer Society Press, 1983

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

[1]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

[1]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[1]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering, August 1985

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[1]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[3]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.