Methodology for Object-Oriented Real-time Systems Analysis and Design

NASA Grant NAG3-1145

Technical Report
Period: October 1990 - October 1991

Software Engineering of Object-Oriented Real-time Systems

by

James D. Schoeffler
Principal Investigator

Department of Computer and
Information Science
Cleveland State University
Cleveland, Ohio

Abstract

Successful application of software engineering methodologies requires an integrated analysis and design life-cycle in which the various phases flow smoothly "seamlessly" from analysis through design to implementation. Furthermore, different analysis methodologies often lead to different structuring of the system so that the transition from analysis to design may be awkward depending on the design methodology to be used. This is especially important when object-oriented programming is to be used for implementation when the original specification and perhaps high-level design is non-object oriented.

In this report, two approaches to real-time systems analysis which can lead to an object-oriented design are contrasted: first, modelling the system using structured analysis with real-time extensions which emphasizes data and control flows followed by the abstraction of objects where the operations or methods of the objects correspond to processes in the data flow diagrams and then design in terms of these objects; and second, modelling the system from the beginning as a set of naturally occurring concurrent entities (objects) each having its own time-behaviour defined by a set of states and state-transition rules and seamlessly transforming the analysis models into high-level design models.

A new concept of a "real-time systems-analysis object" is introduced and becomes the basic building block of a series of seamlessly-connected models which progress from the

system analysis logical models through the physical architectural models and the high-level design stages. The methodology is appropriate to the overall specification including hardware and software modules. In software modules, the systems analysis objects are transformed into software objects.

1. Introduction

Successful application of software engineering methodologies requires an integrated life-cycle in which the various phases flow smoothly from one to another so that backwards and forwards traceability is straightforward. Jacobson emphasizes the need for "seamless models" to avoid errors defining this concept as: "two models are said to be seamlessly related to one another if concepts introduced in one of the models can be found in the other model through a simple mapping" [Jacobson,87].

There is considerable difference of opinion about integrated life-cycles for the software engineering of real-time systems. The most widely used approach is based upon structured analysis and structured design with real-time extensions [Hatley,88], [Ward,86] and [Bruyn,88]. Another approach views system development as a set of transformations starting from the requirements model and ending with the program model [Jacobson,87]. Jacobson .

More recently, there has been much interest in using object-oriented programming for the implementation of the software aspects of the system design. Meyer suggests that object-oriented programming is the most promising approach to generating reusable software. [Meyer,87]. For a general discussion of object concepts, see [King,89], [Nierstrasz,89].

Meyer makes two especially interesting points. First, the top-down design leads to a structure chart with an excess of data transmission of arguments up and down the structure and he indicates that object-oriented design is the solution to this "tramp data" problem. He then states "the law of inversion": "if there is too much data transmission in your routines, then put your routines into the data". "Instead of building modules around operations, and distributing data structures among the resulting routines, use object-oriented design which does the inverse by attaching the routines to the data structures to which they apply".

Meyer's second point is the need for inheritance to defer features. In particular, he notes that the objects can only be partially defined in the sense that some operations will apply to all instances but that many can be defined as to

their general function but not in detail. He uses an object called "state" to encapsulate the state-operation of an example user interface and notes that operations relating to changing from one state to an other can be completely defined but a state display operation, while clear as to its purpose, must necessarily be dependent upon the particular state being displayed. He indicates that inheritance solves this difficulty by defining such operations to be "virtual" in the parent class and then using sub-classes for each kind of object which requires a different display operation. Thus some operations are deferred in a manner not unlike the deferring of detail in standard top-down design.

Parnas emphasises the importance of "data hiding" in successful designs, suggesting that systems details that are likely to change independently should be the "secrets" of separate modules [Parnas,85]. Data hiding is an important characteristics of objects. It is necessary then to evaluate the software engineering methodology so that a "seamless" boundary exists between the systems analysis phases and the design phases if the design is to be object-oriented.

Falk emphasizes two points: , that the different analysis methodologies have different starting points for the modelling of the system which probably leads to different structuring of the system; and second, that the transition from analysis to design may be awkward depending on the design methodology to be used.

In a later paper, Meyer [Meyer,89] argues that the "bottom up" technique is the real engineering approach and is much more likely to be successful. He argues that bottom-up design is the very idea of reusability. He states that the object-oriented approach to design is a "bottom up" approach and that its main contribution is to tackle head-on the key issues of modular design.

Booch emphasizes the limitations of functional decomposition methods and stresses the advantages of object-oriented development [Booch,86 and 87],[Meyer,87 and 89],[Parnas,85],. He recommends that each module in the system denote an object or class of objects from the problem space. Abstraction and information hiding form the foundation of this object-oriented development.

Earler, Booch indicates that systems defined this way tend to exhibit characteristics quite different than those designed with more traditional functional approaches [Booch,86]. In particular, they tend to be built in layers of abstraction, where each layer denotes a collection of objects and classes of objects with restricted visibility to other layers. He calls such a layer a "subsystem". He also indicates that the global flow of control in an object-oriented system is quite different from that of a

functionally decomposed system which usually has a single thread of control. Rather, he says, object-oriented designs lead to multiple threads of control. The author does not develop this idea further.

[Bailin,89]Bailin author points out that an analysis methodology based on structured analysis methods (even with real-time extensions) does not result in a specification which can be designed in the form of objects (at least easily). He points out that structured analysis methods groups functions together if they are constituent steps in the execution of a higher level function. However these functions may operate on entirely unrelated and different data abstractions. In an object-oriented design, however, the functions represent "methods" of the object and they operate on the data abstraction of the object itself. Consequently, the structured analysis methodology results in a grouping of functions which are associated with different objects. To perform object-oriented design, then, it is necessary to manipulate the results of the structured analysis so that the modules produced correspond to objects. The author indicates that this is very difficult and an undesirable but necessary step. Hence he proposes to constrain the original analysis to the requirement that the result be compatible with object-oriented design.

Although there is general agreement about the object-oriented design and programming methodology once objects have been selected, it is clear that the selection of objects is the most critical and difficult part of the overall system design. From a system perspective, objects must be grouped into tasks and packages (in the Ada nomenclature) [Buhr,84],[Nielson,88]. Consequently, the overall design problem becomes one of determining a structure of the software involving objects, tasks and grouping of objects and tasks including definition of their concurrency characteristics, intertask communication, and scheduling which meets the system specification.

There is, however, no single generally accepted systems analysis methodology which leads seamlessly and naturally to an object-oriented programming implementation for real-time systems. Rather there are two approaches to systems analysis which can lead to an object-oriented design:

1. Model the system using structured analysis with real-time extensions which emphasizes data and control flows. The resulting leveled data and control flow diagrams are then used to abstract objects where the operations or methods of the objects correspond to processes in the data flow diagrams. Real-time aspects are supported during

the analysis phase through states and state-transition descriptions of control processes.

2. Model the system as a set of naturally occurring entities each having a life-cycle of its own defined by a set of states and state-transition rules. The entities become objects. Data flow diagrams are used to model the processing in each state and the relationships among objects. These in turn lead to the definition of operations or methods for each object. Real-time aspects are related to concurrent access of the data hidded in the entities and is left to the design phase.

We conclude that an object-oriented systems analysis is highly desirable but note that it must be applicable at all phases and not merely at the software level. To this end, we introduce the new concept of a "real-time systems-analysis object" which becomes the basic building block of a series of seamlessly-connected models which progress from the system analysis logical models through the physical architectural models and the real-time software models of those portions of the system implemented as software in computers. Because of the concentration on objects from the beginning, there need be no separate transformation to create the objects at the real-time software design and object-oriented programming phases. It is shown that "real-time objects" suitable for this modeling must have many of the characteristics we associate with tasks in a software systems.

We propose an analysis-design life-cycle using these real-time objects which is closely patterned after the conventional structured analysis approaches of Hatley [Hatley,88] , Ward [Ward,86] , and ESML [Bruyn,88]. The combination of these approaches is first summarized in section 2. Section 3 defines real-time objects and their properties. Section 4 presents the life-cycle as a series of seamlessly connected systems analysis models which culminate in an object-oriented software design suitable for an object-oriented programming implementation.

2. Real-time structured analysis plus object-oriented design

The systems analysis and design of real-time systems practiced today is based upon structured analysis with real-time extensions. Much of this work is credited to three authors. Derek Hatley extended the structured specification methodology to real-time systems and successfully applied his methodology in the aerospace environment[Hatley,88]. The so-called real-time extensions added control and timing

considerations to the data flow diagram specification which had previously studiously avoided all questions of "how" or "when". The result is a specification that is much more qualitative but still not formal.

His methodology produces a requirements model which is a "logical" or "essential" model of the system. It is not concerned with hardware or implementation. Processes execute in zero time to avoid concurrency. The requirements model is then transformed into an architecture model which represents the higher levels of hardware and software design.

Ward proposed an extension to system specifications based on data flow diagrams which allows the depiction of a system as a network of potentially concurrent "centers of activity" (transformations), and of data repositories (data stores), linked by communication paths (flows) [Ward,86]. This allows the representation of control and timing in a system specification. The paper is important not only because it is the basis of a commonly used real-time system specification methodology (the so-called Ward-Mellor methodology) but because it introduces the concept of qualitative evaluation of specification including both essential and implementation schema. The author calls the specification a "transformation schema" with the word transformation meaning a process in the sense of conventional structured analysis but extended to allow both data and control processes.

ESML is a combination of the real-time system modeling methodologies of Ward-Mellor and Derek Hatley which is equally applicable to all three of the common approaches to specifying a system: functional hierarchy; event-response; and object-oriented [Bruyn,88]. In addition to combining the ideas of the two methodologies, ESML attempts to make the model much more rigorous so that the specification becomes more quantitative.

We summarize these closely related methods below.

2.1. The requirements model

A requirements specification must be a model of the system in the sense that applying inputs to the model should specify the corresponding outputs. That is, to test whether or not a design or implementation satisfies the specification, it is necessary to be able to test the design or implementation and determine whether it produces the same outputs as the specification. Applying inputs to a model to determine outputs is called "executing" the model.

Non-real-time structured analysis creates a requirements model using data flow diagrams and a data dictionary. the model contains:

1. data stores which are repositories of persistent data. A data store has defined data contents. Data read from a data store does not remove the data from the store.
   Data stores contain no state data. Hence their outputs are a function of their inputs at the time they are triggered and not upon their past history. They are assumed to carry out their transformation in zero time after being triggered. A non-depletable store is a normal data store which contains a defined contents which is persistent and may be read and re-read at will. A depletable store reduces the count of its contents each time it is read and useful for modeling energy or resource usage.

2. control stores which are repositories of state data which is persistent over time and a state machine. Control transforms process input data and control data processes by signalling them or by activating or deactivating them. A data process ignores its inputs when deactivated. An activated data process is triggered when inputs arrive. A signaled data process is one which can produce output at any time because it does not have to wait for data to flow to it along its data flow inputs. Such a process is triggered by a control signal which then causes it to response once to the signal and produce its outputs. Control stores do contain state data and also contain a state machine. They also carry out their transformation of control inputs to ouputs in zero time. They are triggered by arrival of control inputs.

   Buffers are data stores with a specific capacity and limited to a single data flow input and data flow output. An arriving flow adds one unit to the buffer and a flow from the buffer deletes one unit from the buffer. The author mentions that inputs to the buffer when it is at its maximum capacity are lost. He does not defend this strange definition. The author does not mention reading a buffer that is empty. Note that a data store, when read, does not remove the information in the store as does a buffer.

3. data flows which are pipelines along which packets of data of known composition may flow. By data is meant problem oriented data and not control information. Data flows may be discrete or continuous and are represented differently (solid arrow with single or double arrowhead). Flows may be value bearing (data flows whose data contents are either continuously available

or available only at discrete instants of time) or non-value bearing (a signal that an event has occurred).

4. control or event flows are represented by a dotted line with three variations: a control signal has a single arrow head; an activation control flow has a double arrowhead pointing toward the process to be activated; a deactivation control flow has a double arrowhead at the process to be deactivated but pointing backward. The addition of control flows which are defined as pipelines along which control data or no data flow. When no data flows, the control flow acts like a signal or interrupt.

   Prompts represent control imposed byone control transformation on another transformation. Prompts are more extensive than either Hatley or Ward-Mellor methodologies support. Prompts are:

   1. disable/enable which make a transformation (control or data) active or inactive. A process made inactive loses all intermitant results and restarts from scratch when activated again. Since data transformations carry out their actions in zero time and have no state, they are merely activated or deactivated by the disable/enable prompts. Control transformations however do have an internal state. Disable followed by enable of a control process causes it to restart in its initial state.
   2. suspend/resume which makes a process inactive but without loss of state information so that resuming the process causes it to continue from where it left off. Only control transformations have state so the suspend/resume is relevant to them only. Suspend/resue and disable/enable are identical for data transformations which do not have a state.
   3. trigger which causes a data transaction to perform a discrete time action.

5. external entities which are undefined boundaries of the system being specified and which as sources and sinks of data flows entering and leaving the system. External entities and data flows entering and leaving them precisely define the boundaries of the system.

6. data processes which are stateless transformations of input data arriving in data flows entering the process or data read upon demand from a data store. The results of the transformation are output as data flows to data stores, external entities, or other processes. Flow transformations may be primitive (not decomposed further) or non-primitive (hierarchically decomposed). Primitive flow transformations must be unambiguously

defined so that outputs can be calculated from inputs. The form of the specification is not fixed by ESML.

7. control processes (control specs) which are somewhat equivalent to a control process which accepts control flow inputs and produces both control flow outputs and also sets activation/deactivation of processes. Control specs may contain a state (internal data). Control transformations are all primitive and each must be specified. If the control transformation has no state, a process activation table can be used. If the transformation contains a state, a state-transition specification must be provided.

   Control specs are always primitive, that is, not decomposed and precisely defined by either a state transition diagram (if they have a state) or by combinational logic and are denoted by a heavy line (the "bar") on data flow diagrams. Control flows enter and leave the bar.

For large systems, it is convenient to build the scheme in a hierarcy where processes (control and data) at one level are further decomposed or detailed at the next level done. Transformation schema may be hierarchically decomposed in the same way that data flow diagrams in structured specifications are decomposed. Because of the introduction of control processes with internal state and finite state machine, additional rules must be imposed. For example, if a parent process is deactivated, all its childrun processes are deactivated also.

The model is leveled. The highest level is shown on the context diagram which contains one process representing the system, all the external entities, and all data flows to and from the external entities. All these flows enter or leave the system.

The next level decomposes the context process showing the basic processes and data stores of the system and the data flows into and out of them. Each process may be further decomposed into its own data flow diagram which shows the multiple processes into which it is decomposed and the data flows among them. Processes which are not further decomposed are called "primitive" and their transformation must be defined precisely.

A data dictionary collects all definitions of primitive processes, data flows and their contents, and data stores and their contents. The specification then, essentially, consists of the data flow diagrams and the data dictionary.

Integrity of the model implies that data flows entering and leaving one level must appear on the level above. The

leveling is a convenient notation and presentation only. Conceptually, a single data flow diagram showing primitive processes only could be drawn.

2.2. The hardware architecture model

Standard structured analysis proceeded toward high level design by enhancing the logical model. That is, as physical considerations or design decisions were made, they could be incorporated into the specification, there transforming the logical model into a physical model. For example, the decision to use a certain kind of communication line to bring input data into the system could be modelled with the appropriate processes and data stores which are necessary in such devices. Then decisions were made concerning which processes were to be implemented in hardware and which in software. The physical model would be further enhanced to handle new interfaces and other considerations due to these decisions.

The progression to the physical model and its interface to the design of software (tasks and modules) was relatively vague in standard structured design. Hatley enhanced this process with his so-called architecture model.

The transformation of the requirement model to a more physical model including specification of both hardware and software modules and the enhancement of the requirement model using the Hatley notation and approach.

Hatley's architecture model consists of architecture modules, architecture flows, and an interconnect specification. An architecture module is simply a boundary within which are those processes of the data flow diagram assigned to it. Connecting architecture modules are information flow channels (also hardware). Flows into and out of architecture modules must flow across information channels. The advantage of the information channels is that they are precisely defined and localize all information about their capacity and throughput and permit performance evaluation of the design to be done.

The creation of the hardware architecture is done using the leveled data and control flow diagrams. Starting with the top level, the processes are assigned to architecture modules which are then successively decomposed as lower levels are considered. At each stage, the only decision is what modules are to be used and what processes are to be assigned to each. Of course, hardware decisions usually imply enhancement of the model to account for transformations of data from one physical form to another so it can actually enter or leave a hardware module.

A hardware module which is a processor implies that the processes within it are to be implemented in software. The Hatley book does not suggest how this is to be done in the case multiple tasks, real-time operating systems, or real-time languages such as Ada are to be used. But note that a decision to use a distributed system (multiple processors) does indeed specify which processes are to be carried out in which processor and all details of the communication between them. For example, an interconnection through messages on a local area network would entail specification of the local area network, the message passing protocol, and all interfaces between the network and the processor. Only the design of the tasks and software modules and their interconnection within each processor would be left to carry out.

Hatley's architecture model is actually a direct extension of the structured analysis model and as such is a straightforward transformation from it. The key components are:

1. Architecture Flow Diagram -- a diagram showing hardware modules which contain bubbles allocated to that module, data flows from one to another, and data flows cross the boundary of the hardware module. Specification of a hardwre module is simply the specification of the processes in the logical model allocated to it.
2 Leveled Architecture Flow Diagram -- decomposition of hardware modules into submodules.
3. Architecture Interconnect Diagram -- a diagram showing the hardware modules, the physical interconnections between modules, and allocating data flows to the interconnection.
4. Information Flow Vectors -- specification of logical data flows flowing along an interconnection.
5. Physical Model Enhancement by
   -- addition of bubbles to a hardware module for additional processing necessary to accomodate the physical form of interconnection of input and output flows.
   -- additional bubbles defining the physical implementation of a logical data flow

The series of incremental enhancements of the physical model represents a gradual transition from the logical to the physical model. Furthermore, it is possible to think of the high level design process to consist of a succession of architectural models each of which is a simple transformation from the previous. This facilitates considering alternative design decisions. It also means that the design process is more integrally (seamlessly) linked

with the specification. Consequently, it is easy to trace forward and backward between any stages of the specification-design.

2.3. The software architecture model

Each computer in the hardware architecture model has been specified by a set of interconnected data and control processes, data flows, and data stores. Because the hardware architecture has included computers, it has been enhanced so that the interface to the computer has been considered and flows crossing the boundary have been specified in terms of messages or signals on buses. It is next necessary to design a software architecture which consists of a set of interacting real-time tasks which may be implemented with object-oriented programming.

There are several approaches to this problem all of which involve the identification of objects from the structured anlaysis specification [Bailin,89] [Coad,90] [Nielsen,88]. The Nielsen approach is thoroughly docmented and described here.

The objective of the software architecture Model is to recognize concurrency problems, and organize the system into a set of interacting real-time tasks which meet the specification despite the concurrency problems. Of course, throughput and response time specifications must also be met.

The design is based upon virtual machines and objects. Starting from the data flow diagram oriented architectural model of the software, a set of communicating sequential processes are identified by identifying concurrency in the data flow diagram, considers a process to be a task, groups tasks into Ada packages, and designs task bodies. Complex task bodies are further modularized with the objective of data-hiding which leads to implementation as objects.

For a real-time system, this machines contains a set of communicating sequential processes. The processes execute in parallel but each represents a single-thread sequential action.

The first step is called "process abstraction" which involves examining the top level data flow diagram and identifying those bubbles into groups which can be carried out in parallel. Heuristics for doing this are:

Group bubbles associated with an external device.
Group bubbles which have functional cohesion.

Separate bubbles which have time-critical functions so that they will have their own task.
Separate periodic bubbles for scheduling purposes.
Separate non-critical computationally heavy bubbles so that they can be assigned to background tasks.
Group bubbles which have temporal cohesion, that is, have actions which must be carried out at the same time.
Group bubbles whose storage requirements may require secondary storage.
Group bubbles that access a shared data base so that mutual exclusion can be implemented.

It is next necessary to associate the operations with objects by defining the data structures on which the operations act. For example, all the operations of a given process may operate on the same data structure. Then the object becomes that data structure and that set of operations.

Operations may be quite complex and operate on data specific to that operation only or be further decomposed on the data flow diagram. In these cases, decomposition continues by defining this operation to be part of a sub-object containing the data operated on by that operation. This decomposition produce sub-objects rather than sub-processes.

The major principle used in choosing the objects is data-hiding, defining the data structures and defining operations on the data structures which hide the structures from the users.

Decomposition continues until all operations are associated with objects and no further decomposition of any operation is desired.

At this point, the design consists of a set of communicating sequential processes each of which includes one or more objects or hierarchies of objects. Notice that there is no concurrency problem for the objects within a single task. If objects are shared by two processes, however, there is a concurrency problem since processes operate in parallel. In this case, it may be necessary to add additional objects to control this concurrency (eg, monitors or buffers).

Implementation of the design may now proceed using object-oriented programming. The task structure and the interprocess communication of the supporting real-time oeprating system determine how methods of an object in one task are used by an object in another task and how parameters are passed.

2.4. Execution of the model

Any specification must be a model of the system in the sense that applying inputs to the model should specify the corresponding outputs. That is, to test whether or not a design or implementation satisfies the specification, it is necessary to be able to test the design or implementation and determine whether it produces the same outputs as the specification. Applying inputs to a model to determine outputs is called "executing" the model.

A logical or essential model is one in which the system is assumed to be implemented as virtual machines with infinite resources. Execution of a process in zero time means that the process changes state in zero time. Hence the lifetime of a process consists of a series of state changes or executions separated by time interevals during which the state does not change. This is the same concept behind discrete-event simulation of real-time systems and provides a sound basis for evaluation and use of specifications.

The importance of the (discrete-event) model is that responses due to inputs which arrive close together are predicted in an orderly manner. As a practical consideration, there are no concurrency problems such as would arise when two processes both read and update an item in a data store and when the two processes execute in parallel in an implementation, the overlap of their execution may produce erroneous results. This is always a problem in software implementations since task switching might occur at arbitrary instants of time due to an interrupt or other event. Consequently, designs must include mechanisms to prevent concurrency problems (such as preventing concurrent use of data items with semaphores, assigning guard tasks or monitors to data, etc). Thus the model permits the testing of the specification without regard to concurrency problems.

Arrival of two inputs at the same instant may result in a "race" condition in the model. If this is significant in the model, the model must specify a resolution to such conditions. Such race conditions become more important as the model progresses toward a more physical model such as in the hardware and software architectural models. The resolution becomes part of the design of the system. Notice that the successive models incorporate more and more physical considerations which are often considered to be design considerations. Thus the use of successive models blurs the distinction between analysis and design.

The software architecture described above requires the identification of objects as pointed out by Coad [Coad,90].

This step is critical to the success of the system design and appears only at the software architecture stage. A criticism of the approach is that this critical step is very difficult to carry out from the structured analysis specification and results in a break in the series of seamless models comparable to that experienced when using structured design to create the software architecture [Sanden,85 and 88a], [Seidewitz,86a and b], [Bailin,89].

3. Real-time structured analysis and design for object-oriented programming

Our objective is to provide a seamless analysis and design methodology which proceeds from the requirements specification through the actual design phases based entirely on the use of objects as the basic building blocks in order to facilitate the transformation from specification to design to implementation of software using object-oriented programming.

3.1. Structured analysis in terms of objects

We take the point of view that a requirements specification is a strict, although non-formal, representation of the system in the sense that it can be tested to determine the required response to any sequence of inputs and events. Hence any design or implementation can be tested and the results cmpared to those of the requirements specification. In principle, the requirements specification must be sufficient to simulate the system response to inputs and events. Hence an object-oriented specification must be such that a conceptual simulation of all objects, their interactions, input events, and data flows taking place in parallel in real-time yields the required response of the system over time.

The specification of a software system and in some cases a non-software system has been discussed by several authors [Cameron,89], [Bailin,89]. [Coad,90], [Jacobson,87], [Sanden,85,88a,88b,89a,89b], [and [Shlaer,88 and 89]. We summarize from Bailin.

Bailin addresses the specification of a software system starting immediately from an object-oriented point of view [Bailin,89]. He uses data flow to link entities (objects) via "calls". He proposes a top-down approach, decomposing entities into simpler entities resulting in a set of entities which can be implemented using object-oriented programming.

Bailin differentiates between active and passive entities. Active entities are diagram nodes but passive entities appear either as data flows or data stores. A passive entity

corresponds to an entity whose state does not evolve over time and hence is essentially a data abstraction. Its methods or functions operate on the data and are initiated by other entities. Thus an entity is an object, and thus the system is structured from the beginning as a set of objects.

Bailin allows entities to be decomposed into sub-entities. Functions are decomposed just as in DFD specifications. The resulting EDFD hierarchy consists of an upper hierarchy of entities and sub-entities, a wavy line of lowest-level entities, and a hierarchy of functions and sub-functions below each lowest-level entity.

Bailin's object-oriented specification then consists of a hierarchy of EDFDs and a set of entity relationship (ER) diagrams. The ER model shows explicitly the relationships among entities (active and passive). The methodology then proceeds through the following steps:

1. identify key problem-domain entities.
2. distinguish between active and passive entities.
3. establish data flow between active entities.
4. decompose entities and functions into sub-entities and functions.
5. check for new entities
6. group functions under new entities.
7. assign new entities to appropriate domains.

Steps 4 through 7 are iterated to get sufficient detail in the specification. Clearly the method allows the definition of objects suitable for object-oriented programming.

Bailin indicates that the methodology can be used for overall system specifications including hardware but addresses this only by indicating that "entities are allocated to hardware" . He does not give any details of this process. In particular, his concepts of data structures and calleable functions are not mentioned or defined for a hardware entity.

There are some problems with his approach to using objects for systems analysis of general systems as opposed to software systems. They are:

1. The concept of a "call" of a method is not appropriate especially for a pair of communicating hardware objects. Even for software objects, this is not appropriate for objects which end up in separate tasks where "triggering" would be more appropriate.
2. At the specification level, various objects exist and operate conceptually in parallel. Hence concurrency, blocking, and collision must be

defined and understood if the specification is to actually represent a model of the system.

3. He makes no distinction between class and instance of the class.

Our objective is a more general object-oriented systems analysis procedure which can be used throughout the analysis project, uses objects throughout so that software objects evolve naturally from the analysis, and makes objects created during the analysis as reusable as possible in similar applications. To this end, we adopt as many of the concepts of both structured analysis and design and object oriented analysis as described above as possible in order to take advantage of techniques which have been demonstrated as successful and useful in specification writing. We emhasize those aspects which are different in the discussions below.

In the next section we define a "systems analysis object" which can be used for the building of a specification or model of a system at both the essential and physical levels, and which leads naturally to software objects for those portions of the system implemented in software.

3.2. Definition of systems analysis objects

We first define a Systems Analysis Object which is the basic building block of object-oriented systems analysis. It differs from a software object but is transformable to a software object for those sections of a design which are implemented as a real-time software system. The guiding requirement is that an analysis is a model of a system which we conceptually examine through discrete event simulation of the model.

We use the following terminology for clarity. An object is a specific instance of a class which in turn is a template definition common to all instances of that class . Our model consists of a set of concurrent systems analysis object instances of classes defined within the model.

A systems analysis object class is similar to the common notion of a software object but different in important ways. A class defines an entity containing:

1. an optional state -- a set of attributes (variables) which define the response of the object instances of the class to events. For example, an object representing a military plane might have a state which indicates whether the airplane is in the cruise or attack mode.

2. application-specific attributes (variables) which may be private to the class or public to the class. Private

attributes are accessible only to the processes (methods) of the class, Public attributes are directly accessible to the processes of any class. The attributes contain data values relevant to the specific class and are problem-oriented. They are differentiated from the state attributes only because the understanding of the class is so dependent upon the notion of state that it is worthwhile emphasizing this concept. The inclusion of public and private attributes is done in the spirit of data-hiding and for the purpose of creating re-usable systems anlaysis objects

3. processes (methods) which are triggered by a data-flow (defined below). A process is a transformation of the object-instance's attributes and state which executes concurrently with other processes of this and other objects over time but in the sense of discrete event simulation. Two different views of processes are commonly used in discrete event simulations: the "time-process" view and the "event" view.

   In the "event" view, a triggered process is a function which may change the state and attributes of its object instance and may trigger other processes of this object instance or other object instances all at the instant in time at which the process is triggered. An example of an "event" process might be the process triggered by the pilot of the above airplane signalling a change from cruise to attack modes. The function of the process would change the state accordingly and enable and disable other processes of the airplane object instance (see below).

   In the "time-process" view, a triggered process makes a series of states changes, each at a discrete instant of time. In this view of a process, one imagines that the state and attributes of the object instance are not changed by the process between these discrete times instants at which the state is changed. One may view the process as an "event" process which after the initial triggering, is triggered again at later time instants through internal rather than external controls. Once triggered, a "time-process" ignores any further triggers because it controls internally the future triggers which cause state changes. Thus the evolution over time of such a process is defined by the process specification itself. Since the process does nothing between state changes, the time interval between changes is specified with the Simula statements: "wait T second" and "wait until some-event". Hence the specification of the process is simply a function containing these statements as well as the usual structured englished statements used to

specificy a transformation. The effect of these statements is to insert a time period between the succesive state changes specified via the structured english.

A simple example of a "time-process" might be the process which "arms" some missles. This process is triggered by a pilot command. The arming of the missles takes a finite and significant amount of time. Validity of the model must take this time in account so that for example, a firing event cannot occur before the missles are finished being armed. This process might have the structured english specification:

```
disable external triggering of this process
set missle-state to "arming in process"
wait 7 seconds
set missle-state to "armed"
trigger missle-armed process
```

This specification contains two state changes 7 seconds apart each of which takes place in zero time. Note that the first state change prevents external triggering of this process which means that arrival of an external trigger would simply be ignored according to this specification.

It is well known that any dynamic system can be modeled by either the "event" or "time-process" views. Our objective is to produce a specification of a system and clarity of that specification is paramount (as is its completeness and testability). By allowing both types of processes within systems analysis objects, we gain ease of understanding the specification.

We note that the discrete event simulation concept of state changes in zero-time at discrete time instants provides a straightforward way to resolve concurrency problems among object-instances within a specification. Any concurrency problems due to state changes within a single object instance or in two or more object instances at the same time instant are imagined to take place in some sequence even though they take zero time and are all completed at that time instant. The specification must explicitly recognize and resolve any race conditions or ambiguities which might arise when this happens. Furthermore, there is no implict queueing of data flow triggers and their accompanying data. The flow of a trigger to a process immediately triggers the process or is totaly ignored. If the specification is describing a system in which queueing of data triggers are necessary, this queueing must be explicitly modeled.

4. enable/disable states of all processes (methods). An implicit state of the object-instance is always present. This state indicates for each process whether it is enabled or disabled. An enabled process responds to an arriving data-flow trigger. A disabled process ignores any arriving data-flow trigger. A process (method) of an object-instance may enable or disable any other process of the same object-instance but is prevented from enabling or disabling a process of another object-instance. Hence in the specification (model) an object which needs to disable a process of another object instance must trigger a process (method) of that other object instance which in turn will do the disabling.

5. receives and transmits triggers along data flows. A data flow is a defined path along which defined data packed may travel. Thus data may or may not accompany a trigger but unlike the "call" of a software method, no data is returned as part of the trigger. A process (method) which must output data to any process must have a data flow to that process and must initiate a data flow to that process which in turn triggers the process.

6. may inherit state, attributes and processes (methods) from one or more classes. This inheritance implies that this derived object class contains all the public and private attributes of the parent classes as well as additional attributes defined for this class. All the processes (methods) of the parent classes are also inherited. Additional processes may be defined for this class and any inherited processes may be over-ridden or redefined for this class. Inheritance is an important concept in a specification because it permits a class to be defined as a modification of other classes which in turn helps decrease the volume of the specification and increases its understandability. It is particularly important when attempting to reuse specification classes because it allows their specialization to the problem at hand without modification of an already existing class.

3.3. Object data-flow diagrams

Object-oriented design methods have been frequently criticized because of the difficulty of tracing the response to an event whereas this is a positive attribute of structured analysis data-flow diagrams. In a specification, it is important to be able to trace the jresponse to an event and consequently, we emphasize this data flow through object data-flow diagrams on which are shown object instances with relevant (but not necessarily all) processes

identified as part of the object instance and data flows exiting the process which initiates the data flow and enting the target process which is triggered by the data flow. There have been many suggestions for graphical icons which may be used to display an object-instance, its relevant processes (methods), and even public attributes [Wasserman,90]. The data flow icon may be taken to be an arrow from source process to target process. Data which is passed with the data flow to the target may be defined in a data dictionary as in structured analysis methodologies, or be shown with a companion small arrow with a circle at the tail as is done in a structure chart in the structured design methodology. A dash arrow might be used to indicate pure triggering data flows as is done in real-time structured analysis for control flows.

The external entitiy of structured analysis is retained as the source of data flows crossing the system boundary (entering and leaving the context abstract object).

What ever the graphical icons adopted, the object data-flow diagram can then be used as a basis for tracing the response to any event or situation, a major advantage of standard structured analysis specifications.

3.4. Decomposition of analysis objects

It is advantagous to retain the hierarchial structuring of the specification as in standard structured analysis and the Hatley real-time methodology with one exception. The Hatley real-time structured analysis methodology separates control and data processes and allows data processes to be hierarchially decomposed but not control processes. In effect, all control processes are terminal processes but only the lowest level data processes are terminal. We adopt the following decomposition strategy.

Define an abstract object instance as one which itself has no processes (methods), state, or data attributes but does contain object instances. All data flows to this object terminate on processes of the included object instances and all data flows emanating from this object exit from processes of the included object instances. All totally internal data flows are between processes of the included object instances. A abstract object instance has no defined class associated with it. Thus the abstract object instance is simply a conglomeration of object instances for the purpose of simplying the understanding of the specification.

Define a terminal object instance as any non-abstract object instance. It must be an instance of a defined class. It has attributes, state, and processes (methods). The processes may be terminal or abstract.

Define a terminal process as a process of an object class (method) whose transformation is defined explicitly using, for example, structured english with possible extensions for time-processes (wait and wait-until). A terminal process is not further decomposed.

Define an abstract process as a process of an object class (method) which is further decomposed into a set of processes in a hierarchial sense. Data flows entering and leaving the process are the identical data flows entering and leaving the decomposing set of processes.

Our object-oriented specification, whether at the essential model level or the physical model level where many design decisions have been already made, will consist of a set of object-instances interconnected via data-flows. Furthermore, if the model were actually simulated, the object instances simulated would be terminal object instances.

The abstract object instance has the advantage that it is not necessary to find a single "action verb" to describe the object. Hence a system can be modeled by an abstract object which can then be subdivided into subsystems each of which becomes an abstract object. Terminal object instances are introduced at the appropriate level. This retains the hierarchial organization of a structured analysis specification while forcing an object-oriented structuring at every level.

It is convenient to allow a terminal object instance (not its class definition) to contain abstract objects and/or abstract processes. The abstract object or process then simply represents the set of terminal objects and processes in a more compact form for clarity and understanding purposes. Thus the abstract object or processes could be substituted by their decompositions.

The development of the hierarchial object-oriented specification applies decomposition in a variety of different manners as follows.

1. An abstract object instance might be divided into two (abstract or terminal) object instances. All entering and exiting data flows must enter and exit methods of the two new object instances. For example the system as a whole might be specified as a single abstract object (equivalent to a context diagram in structured analysis) and then decomposed in abstract objects which represent subsystems of the total system.

2. An object class might be divided into two separate object classes which trigger one another's methods. Each object instance of the original class would be

replaced by a pair of object instances of the new classes with data flows between them. Each data flow into the original instance would terminate on a method of one of the replacing instances. Each data flow leaving the original instance would leave a method of one of the replacing instances.

3. A process of an object class might be decomposed into several processes with interacting data flows. This is the standard decomposition of structured analysis.

Notice that decomposition methods (1) and (2) lead to a hierarchial relationship among abstract and terminal objects. Decomposition method (3) is a hierarchial decomposition of a single process of an object.

Another form of decomposition which is not hierarchial may naturally occur. Two interacting object instances might be decomposed by grouping some of the attributes and state of each object instance into a new instance along with processes (methods) from each as appropriate. The result would be two redefined object instances and an entirely new object instance with interacting data flows. Notice that this decomposition is not hierarchial. This is better treated as a refinement of objects during the construction of the model and the two objects simply replaced by the three. We do this primarily to preserve the hierarchial relationships among objects.

Note that classes may be defined by deriving them from other classes (that is, using inheritance). This is not a form of decomposition although a central structuring method in a specification. This form is especially useful when deriving reusable classes or using existing classes produced in other specifications.

3.5. Form of the structured object-oriented analysis model

The structured object-oriented specification will consist of:

1. An inheritance diagram or the equivalent showing the inheritance relationships among classes.

2. A specification for each terminal class including its public and private attributes, contained object instances, and processes and their specification.

3. A hierarchial set of object data flow diagrams starting from the context level containing a single abstract object representing the entire system and followed by

successive decompositions of abstract object instances with each decomposition of an abstract object instance consituting a lower level.

The many relationships among objects are all present in one of the above elements of the specification:

1. One object "contains" another: the data portion of a class definition includes an instance of another object class. This is shown in the class definitions.
2. One object "triggers" another: a method of an object class triggers a method of an instance of another object class. This is shown in the object data-flow diagrams.
3. One object class is "derived" from another or "inherits" another class. This is shown in the inheritance diagram.

Of course a separate entity-relationship diagram would be useful in some specificiations, especially if there are many object classes representing a data base with application-dependent relationships among these classes.

4. Structured object-oriented analysis and design methodology

Using the concept of systems analysis objects described above, we now propose the analysis and design phases of a software engineering life-cycle which uses consistent models and whose phases may be incrementally applied resulting in a sequence of seamlessly related models culminating in an object-oriented model suitable for implementation of real-time software using object-oriented programming. The life-cycle does not make a sharp distinction between analysis and design, preferring instead to emphasize later models as being closer to the final physical model which is truly a design in the case of software. We will call this model of the software portion of a system the real-time software architecture. In other design methodologies, this final model might be called "high level design". The principal aim of this methodology is to identify objects early in the analysis phase, repeatedly enhance and redefine them in later phases and end up with objects already specified when software implementation begins. This is in response to the often noted statement that it is very difficult to transform a non-object-oriented specification or high level design into a set of software objects.

4.1. The steps of the methodology

The structured object-oriented analysis and design methodology follows that of the real-time structured analysis life-cycle. The steps are:

1. Create the requirements model

   The requirements model in the form of an object structured specification. This requirement model would be termed an "essential" model or a "logical" model if the system does not yet exist but may be a "physical" model if the system or much of its structure either exists or is constrained. In any case, this specification would consists of a set of interacting object instances and could (at least conceptually) be simulated to determine the required response to any events.

2. Create the architectural model

   Following the lead of the real-time structured analysis life-cycle, the first model would be successively transformed into an architectural model by imposing hardware boundaries with hardware connection paths where appropriate and assigning object-instances from the previous model to these hardware boundaries. Enhancement of the model is often necessary such as:

   1. adding object classes and instances due to hardware decisions such as grouping of data flows along a communication path or changing physical form of a data flow such as adding sensors and A/D converters to acquire data for a hardware computer module.
   2. modifying classes because of splitting of functions among two or more hardware modules.
   3. modifying classes and adding classes because of concurrency or sequencing requirements associated with hardware decisions.

   As in the Hatley methodology, hardware boundaries are introduced which we model as abstract objects. Similarly, data flows crossing these boundaries have to be assigned to hardware channels of some kind. Hence each such channel is modeled as an abstract object.

   A design is viewed as a transformation from a specification to a final physical model perhaps with several intermediate models. We adopt the approach of Hatley which suggests a sequence of transformations each adding hardware/software design decisions and the necessary enhancements to the object model these imply. Hence this step may actually result in a series of models or designs.

3. Create the software architectural model

For each hardware module which corresponded to a computer in the architectural model, a software architecture must be created. This important step is discussed separately in the next section.

4. Transform system-analysis objects into software objects Systems analysis objects are not the same as software objects but are closely related. The final design stage of the analysis-design life-cycle prior to implementation of the software is the changover of objects in the software architecture model to software objects suitable for object-oriented programming. This step is discussed separately below.

4.2. The real-time object-oriented software architectural model

Each hardware module in the final hardware architectural model which has been selected to be a computer must further be enhanced to specify the hardware architecture. Consider one such computer module. At this stage, certain object instances have been assigned to the computer. Furthermore, all data flows crossing the hardware boundaries have been specified (through enhancements at earlier stages) as to the hardware channel across which they travel (eg, standard I/O operations to other hardware modules which represent devices, multiplexors, sensors, A/D converters, etc) and the structured english specification of the terminal processes (methods) of objects within the module refer to these paths. However, the objects are still systems analysis objects and considered to execute concurrently with their stage changes taking place in zero time. Note that although we are still working with transformations of the same type of model as used in earlier stages, this level is actually a high level design level in most software engineering life-cycles. We will, however, continue to call it specification to emphasize the seamless nature of the models we use.

The process of changing the systems analysis objects into a structure which is amenable to real-time software implementation takes advantage of the fact that the model is entirely object-oriented and hence there is no special step required to identify software objects. We will continue, however, to enhance the model during these later specification stages which may introduce additional objects.

Real-time software can be thought of as a set of interacting tasks executing under the control of a real-time operating system. The real-time operating system may be implicit (as in an Ada environment where scheduling and other task-control statements are part of the implementation language) or explicit. In the latter case, scheduling and other task-control statements are not part of the implementation language but rather are calls to systems services.

Furthermore, the real-time software is very much dependent on the specific facilities offered by the real-time operating system (eg, dynamic control over task priority level, etc). Nonetheless, the most critical areas of real-time software design are more general than this and include:

1. Identification and resolution of concurrency situations
2. Design for adequate response time and throughput
3. Design for adequate error detection and recovery

Concurrency situations arise from access to shared data and shared modules. Concurrency has already been considered earlier in the models where public and private data of an object instance required access control over a period of time. As object instances are grouped into tasks, inter-task communication is required, and this may significantly affect concurrency, response time, and throughput. Error recovery especially requires enhancement of the model to insure that adequate data is somehow logged or retained to permit recovery. These enhancements further affect response time and throughput and introduce concurrency situations (eg, transaction logging prior to commit, etc).

The real-time software architectural model consists of a set of interacting tasks. Each task is represented by a boundary surrounding certain object-instances. Each task is imagined to be implemented by a single software module. Hence a task module may contain one or more object instances. Any data flows crossing the task boundary must be realized through inter-task communication. The data flows among the systems analysis object instances also imply triggering of the destination process. Within the task, however, the triggering of software modules realizing processes of the object instances must be done through software procedure calls or the equivalent. It is the task itself which is triggered. Hence the triggering of process modules by data flows and the transfer of data accompanying a data flow must be separately considered.

An object instance's access of public data of another instance has been treated like a simple look-up or read of the data because no process had to be triggered. In the software environment, however, this access has to be specified more completely because the method by which it is carried out may seriously affect concurrency, response time, throughput, and error recovery. For example, an object-instance may contain public data which is shared among many objects resulting in many tasks sharing the data in the software model. The object containing the data may be specified to be a shared module permanently memory-resident at a known location as opposed to a task. Hence tasks could directly reference the public data or call the processes (methods) of this object. This would be equivalent to a global common data area. Concurrency problems and errors for

such designs have to be carefully considered in choosing the objects assigned to tasks which will then access such an area.

In any case, the software architecture model is derived from the hardware architecture model by the following steps:

1. Assign object instances to tasks. This packaging step is based upon the amount of interaction among objects, objects whose methods must be carried out at the same time or at the same rate, etc.

2. Specify how the data flows between tasks are to be implemented via inter-task communication facilities of the real-time operating system. This may require enhancement of the object instances including both modifications to processes and addition of objects.

3. Specify how a task is to be triggered or scheduled. For example, the task may execute periodically, upon demand of any other task, or upon arrival of a message from another task through an inter-task communication facility such as a mailbox. Extensive work has been done on real-time scheduling [Sha,90].

4. Examine the concurrency among the tasks and resolve any problems using enhancements to the model (eg, addition of a monitor to control access to public data), addition of concurrency conventions (eg, introduction of semaphores),

It is at the software architectural level that the most critical design is done. The software architecture must be evaluated in the three critical areas listed above. The model is still "executable" in the sense that this model can be simulated as before. A primary purpose of the simulation is to detect concurrency situations which might invalidate the design when the tasks execute processes in other than zero-time.

Prototyping of the objects and tasks at the software architectural level also provides a means of testing response time and throughput as well as detecting concurrency problems.

4.3. Transformation of systems analysis objects into software objects

Upon completion of the software architectural model, individual tasks contain systems analysis as opposed to software objects. Systems analysis objects and software objects are not the same although they are similar, For

example, processes of objects communicate with one another by triggering the destination process and passing data at the same time. At the physical level in software, the communication may be a call if both the source and desination are in the same software module, perhaps a remote procedure call if they are in different hardware modules on a network, global references to data, or communication via inter-task message passing.

A task is represented as a single program. Hence the next design step is to transform the systems analysis objects within one task into software objects within one program. This involves the following conceptual items:

1. Triggers versus procedure calls. Each data flow may be replaced by a call of a method of the destination object with the data transferred as arguments of the call. This requires enhancement of the object class definition to turn it into a software object declaration appropriate to the implementation language.
2. Enhancement of methods to permit data return to the source method. Recall that systems analysis objects permit data transfer only in one direction in order to preserve the ability to track the response to an event. BBecause software objects permit exchange of data in both directions, it is possible to modify the methods of an object and perhaps simplify them. This would clearly be possible if method1 of object instance1 triggers method1 of object instance 2 (and passes it data) which in turn triggers method2 of object instance1 passing it data. Then the two methods of object instance1 might be combined in a single method which calls method1 of object instance2 with both input and output arguments.
3. Explicit scheduling of data flows to control concurrency. Methods may trigger multiple methods of other object instances. When data is being shared, it may be possible to eliminate a concurrency problem with the data by simply controlling the sequence in which the other methods are called. This too results in enhancements to the methods of the software objects. For a general discussion of concurrency in object-oriented programming, see [Nelson,91],[Tomlinson,89].
4. Addition of software-specific objects for control of data structures. within systems analysis objects, data might consists of multiple items where specifc items must be retrieved by one or more identifying keys or by the sequence in which they were added or they must be maintained in certain sequences determined by their data values. In these cases, linked list objects, sorted array objects and other software specific objects commonly called "container classes" might be added to the task's object instances.

Upon completion of this phase of the specification/design, software classes are completely specified and implementation can begin.

4.4. Critical analysis situations

The requirement that a specification be capable of (at least conceptual) simulation implies that the state of an object evolve over time as a series of changes at discrete instants of time separated by time intervals whose duration may be zero. Each state change takes place in zero time.

This poses two severe conceptual problems for real-time system specification: first, objects may at times be unavailable to respond to an input data-flow and yet the specification must yield the correct response to such situations; and second, concurrent data-flows to the object must be possible even in the situation where the object's response is not instantaneous. The first problem requires the object's methods be capable of being enabled or disabled. An enabled method responds when triggered by a data-flow whereas a disabled method ignores the data-flow. The second problem further requires explicit management of concurrent access by including a "monitor" function with the object.

Consider first objects with unavailability intervals. The common problem of "Wait n secs" is such a case because of the need to include a specific time interval before another event can be occur.

> For example, consider the object Antenna representing an antenna on a space vehicle which cannot be used until it has been deployed. The method Antenna_deploy is triggered when the antenna should be readied. But deployment requires a specific time so that other methods of Antenna cannot be used until that time has elapsed. In an entity-life oriented discrete event simulation language such as SIMULA, the method would include the "wait n secs" step. There are two solutions to this problem.
>
> First, object Antenna could include a hidden state variable indicating whether or not the antenna is deployed. This variable would remain at its un-deployed value until changed by, perhaps, an external object which detects successful deployment. Each Antenna method must be specified to test the state variable to determine whether to carry out its response function or to do nothing in response to a triggering data-flow.
>
> Second, the Antenna_deploy method could disable all the other methods which would then ignore any triggering

data flows. The external object which detects successful deployment, would then use an Antenna method to enable its previously disabled methods.

In either case, specification of objects interacting with the Antenna object must understand that the Antenna methods may be disabled or be non-responding at times and include proper specification for situations where the methods do not respond.

Consider the common problem of "wait_until" in which an object must synchronize with another object at an undetermined time defined by a data condition within the second object.

For example, object Scan is periodically triggered to scan a temperature data sensor, compare the resulting value to a limit, and set its internal alarm value variable to OFF or ON depending upon whether or not the limit is exceeded.

Depending upon conditions, method Alarm_recover of object Alarm sometimes should be triggered when the above alarm level goes from OFF to ON. The conditions under which it is to be triggered are internal to object Alarm. Thus object Scan cannot know whether or not to trigger method Alarm_recover. This problem can be handled by decomposing method Alarm_recover into two methods, Alarm_connect and Alarm_handle where Alarm_connect sends a data-flow to a method of object Scan which records the desire to be notified, and Alarm_handle is the method which is subsequently triggered by object Scan when the alarm condition is detected.

An alternate solution adds a state variable to object Alarm indicating interest in responding to the alarm condition. Object Scan can then trigger the alarm recovery method of object Alarm each time the alarm occurs and that method can determine whether or not to respond by examining its internal state variable.

A third solution makes use of enable/disable of the Alarm_recover method itself, still allowing object Scan to trigger the method each time the alarm condition occurs. If object Alarm should not respond, that method is disabled and hence the triggering data flow is ignored. If it should respond, the method is enabled and then responds to the triggering data flow.

Even if methods execute infinitely fast, it is possible to create a method that takes a finite time to execute.

For example: Let Object2 be time triggered each 1 second at which time the value X and its timestamp are updated. The clock triggers method O2_update. Let Object2 have a second method O2_value which returns the value of X and its timestamp.

Let Object 1 have a method which executes:

```
t_stamp = current_time - 1
while (current_time - t_stamp) > 0.25
     (value,t_stamp) = Object2.O2_value();
```

The while loop executes an infinite number of times, producing a wait of up to 0.75 seconds because the loop executes until the next time tick if its first read is more than 0.25 seconds old.

Execution of our object-oriented systems analysis model makes state changes in zero time as in the usual discrete-event simulation models of systems. This leads to a possibly ambiguous result because the triggering of an object can involve triggering of processes of two or more other objects at the same instant. Changes to the sequence of state changes which result at a single instant of time can result in different final states of the system. We note that there is the same possible ambiguity in the Ward-Hatley-EMSL models. Such ambiguity is also present in real systems. For example, the arrival of a set of orders in a mail delivery and the subsequent filling of those orders can result in quite different results depending upon the order in which the orders which were all received at the same time are actually filled. A particular order might be filled if one sequence is used but only partially filled for a different sequence. Clearly the specification of a system must then specify sequence if in fact the sequence is important. In the order filling example, the specification might insert a process (step) which sorts the orders by some criterion (value of the order, age of inventory of ordered items, etc) to remove such ambiguity. In the software area, recognition of possible ambiguities is often important.

For example, suppose that a data-acquisition system is specified to report sensor values on an exception basis (that is, report new samples only if the value has changed more than some specified amount. Suppose it maintains a list of such exception-values as it periodically scans the sensors and detects exceptions. Suppose further that perodically a process collects exceptions, packs them into a message, and transmits the messages across a network.

Then the triggering of the scan process which builds the exception list and the message-packing process which takes exception values in the list and builds a

message for reporting might occur at the same instant. The sequence in which the processes are carried out can result in the message-packing occurring before the scanning so that only exception values previously scanned are packed into a message for transmission. This would result in a delay in transmission of the exception values equal to the time between triggering of scanning if it occurred often and a consequent deteriation of the response time of the design.

Recognition of this should result in its consideration in the design, by assigning the objects owning these processes to different tasks and assigning the scan task higher priority for example, or by creating another object which explicitly triggers both of these processes in the proper order.

5. Conclusions

The two approaches to analysis and design are very similar but differ in one significant respect: the object-oriented approach clusters data and processes into objects from the very beginning. Hence the transition from analysis to design in this approach does not require an abrupt change in the model through the introduction of objects as it does using the structured analysis methodology followed by the Bailin approach discussed in section 3.1 which involves abstracting objects from the data flow diagrams. Although this approach has been proposed and used [Seidewitz,86a and 86b], it seems to be much more difficult to carry through and is subject to more effort if the analysis model ahead of this stage is modified. In contrast, an analysis model already object oriented does not require this step and does not require extra effort to incorporate modifications to the earlier models. This is the payoff of a series of "seamless" models.

There also seems to be a conceptual simplification using the object oriented approach with systems analysis objects as the basic building blocks since such objects correspond to natural systems and subsystems whereas structured analysis requires a single function to be associated with this decomposition.

One potential advantage for the use of systems analysis objects may be in the reuse of portions of an analysis model. For example, an organization which creates multiple applications often finds that the applications interact with one another and this interaction is important in the analysis and design. With an object oriented systems analysis model, the interaction may result in the sharing of objects so that previous analysis objects might be incorporated into the new model. In other situations, the objects might need to be tailored to a new application and

again the use of inheritance may allow the reuse of previously created systems analysis objects.

6. References

[Bailin,89] Bailin, Sidney C., An Object-Oriented Requirements Specification Method, Communication of the ACM, Vol. 32, No. 5, May, 1989, pp. 608-623.

[Booch,86] Booch, G., Object-Oriented Development, IEEE Trans. on Software Engineering, Volume SE-12, No. 2, February, 1986, pp. 211-221.

[Booch,87] Booch, Grady, Software Engineering with Ada, Second Edition, The Benjamin/Cummings Publishing Co, Menlo Park, Calif., 1987

[Bruyn,88] Bruyn, William, Randall Jensen, Dinesh Keskar, Paul Ward, ESML: An Extended Systems Modeling Language Based on the Data Flow Diagram, ACM Sigsoft, Software Engineering Notes, Vol. 13, No. 1, January, 1988, pp 58-67.

[Buhr,84] Buhr, R.J.A., System Design With Ada, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984

[Cameron,89] Cameron, John, JSP & JSD - The Jackson Approach to Software Development (Second Edition), IEEE Computer Society Press, Book No. 858, May, 1989.

[Coad,90] Coad, Peter and Edward Yourdon, Object-Oriented Analysis, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1990

[Hatley,88] Hatley, D and Imtiaz Pirbhai, Strategies for Real-Time System Specification, Dorset House Publishing, NYC, 1988.

[Jacobson,87] Jacobson, Ivar, Object-Oriented Development in an Industrial Environment, OOPSLA '87 Proceedings, ACM Press, October , 1987, pp. 183-191.

[King,89] King, R., My cat is object-oriented, in Object-oriented concepts, databases, and applications, Ed. by Won Kim and Frederick H. Lochovsky, Addison-Wesley, 1989, pp 23-30.

[Meyer,87] Meyer, B., Resusability: The Case for Object-Oriented Design, IEEE Software, March, 1987, pp. 50-64.

[Meyer,89] Meyer, Bertrand, "From Structured Programming to Object-Oriented Design: The Road to Eiffel, Structured Programming, Vol. 1, No. 1, 1989, pp 19-39.

[Nelson,91] Nelson, Michael L., "Concurrency & object-oriented programming", ACM SIGPLAN Notices, Vol. 26, No. 10, Oct, 1991, 63-72.

[Nielsen,88] Nielsen, Kjell and Ken Shumate, Designing Large Real-Time Systems With Ada, Intertext Publications,

Multiscience Press, Inc., McGraw-Hill Book Co, NYC, 1988

[Nierstrasz,89] Nierstrasz, Oscar, A Survey of Object-Oriented Concepts, in Object-oriented concepts, databases, and applications, Ed. by Won Kim and Frederick H. Lochovsky, Addison-Wesley, 1989, pp 3-21.

[Parnas,85] Parnas, David, Paul Clements, and David Weiss, The Modular Structure of Complex Systems, IEEE Trans. on Software Engineering, Vol. SE-11, No. 3, March, 1985, pp 259-266.

[Sanden,85] Sanden, Bo, "An Entity-Life Modeling Approach to the Design of Concurrent Software, Communications of the ACM, Vol. 32, No. 3, March, 1989, 330-343.

[Sanden,88a] Sanden, Bo, "Entity-Life Modeling and Structured Analysis in Real-Time Software Design -- A Comparison, Communications of the ACM, Vol. 32, No. 12, December, 1989, pp 1458-1466.

[Sanden,88b] Sanden, Bo, An Example of Concurrent Software Design in Ada, Report CSSE-88-13, Center for Software Systems Engineering, George Mason University, Fairfax, VA, 1988, pp.1-21.

[Sanden,89a] Sanden, Bo, Concurrent Tasks in Realtime Software Design, Report CSSE-88-03, Center for Software Systems Engineering, George Mason University, Fairfax, VA, 1988.

[Sanden,89b] Sanden, Bo, Systems Programming With JSP: Example -- A VDU Controller, Communications of the ACM, Vol. 28, No. 10, October 1985, 1059-1067. Also Structured Methodology letters, ACM Forum, Communications of the ACM, Vol. 29, No. 2, February, 1986.

[Seidewitz,86a] Seidewitz, Ed and Mike Stark, General Object-Oriented Software Development, Software Engineering Labratory Report SEL-86-002, NASA Goddard Space Flight Center, Greenbelt, MD

[Seidewitz,86b] Seidewitz, Ed and Mike Stark, Towards A General Object-Oriented Software Development Methodology, Proceedings of the 1st International Conference on Ada Applications for the Space Station, June 1986, pp 2-25 to 2-37.

[Sha,90] Sha, Lui and John B. Goodenough, Real-Time Scheduling Theory and Ada, IEEE Computer, Vol. 23, No. 4, April, 1990, pp. 32-52.

[Shlaer,88] Shlaer, Sally and Stephen J. Mellor, An Object-Oriented Approach to Domain Analysis, ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 5, July, 1989, pp 66-77.

[Shlaer,89] Shlaer, Sally and Stephen Mellor, Object-Oriented Systems Analysis, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1988

[Tomlinson,89] Tomlinson, Chris and Mark Scheevel, Concurrent Object-Oriented Programming Languages, in Object-oriented concepts, databases, and applications,

Ed. by Won Kim and Frederick H. Lochovsky, Addison-Wesley, 1989, pp 79-124.

[Ward,86] Ward, Paul T., The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February, 1986, pp. 198-210.

[Wasserman,90] Wasserman, Anthony I., Peter A. Pircher, and Robert J. Muller, The Object-Oriented Structured Design Notation for Software Design Representation, IEEE Computer, March, 1990, pp. 50-62.