

**INTEGRATING END-TO-END THREADS OF CONTROL INTO
OBJECT-ORIENTED ANALYSIS AND DESIGN**

512-61

12694

p. 24

Janet E. McCandlish
TRW
213 Wynn Drive
Huntsville, Ala 35805

Dr. James R. MacDonald
ALPHATECH, Inc.
50 Mall Road
Boston, Mass 01803

Dr. Sara J. Graves
University of Alabama
in Huntsville
Alumni House, Room 102
Huntsville, Ala 35899

ABSTRACT

Current object-oriented analysis and design methodologies fall short in their use of mechanisms for identifying threads of control for the system being developed. The scenarios which typically describe a system are more global than looking at the individual objects and representing their behavior. Unlike conventional methodologies that use data flow and process-dependency diagrams, object-oriented methodologies do not provide a model for representing these global threads end-to-end.

Tracing through threads of control is key to ensuring that a system is complete and timing constraints are addressed. The existence of multiple threads of control in a system necessitates a partitioning of the system into processes. This paper describes the application and representation of end-to-end threads of control to the object-oriented analysis and design process using object-oriented constructs. The issue of representation is viewed as a grouping problem, that is, how to group classes/objects at a higher level of abstraction so that the system may be viewed as a whole with both classes/objects and their associated dynamic behavior. Existing object-oriented development methodology techniques are extended by adding design-level constructs termed logical composite classes and process composite classes. Logical composite classes are design-level classes which group classes/objects both logically and by thread of control information. Process composite classes further refine the logical composite class groupings by using process partitioning criteria to produce optimum concurrent execution results. The goal of these design-level constructs is to ultimately provide the basis for a mechanism that can support the creation of process composite classes in an automated way. Using an automated mechanism makes it easier to partition a system into concurrently executing elements that can be run in parallel on multiple processors.

INTRODUCTION

The philosophy upon which object-oriented analysis and design is based does not lend itself well to the representation of how a system operates as a whole. The object-oriented premise emphasizes the extraction of objects to be modeled from the problem domain in contrast to traditional methods which rely on the functionality of the system. A review of

some of the more current object-oriented analysis and design methodologies highlights this shortcoming by these methodologies' inability to effectively represent the end-to-end processing of a system. A global representation is key to understanding how the system operates. As described in [Fichman], conventional methodologies use tools such as data flow and process-dependency diagrams for representing global threads end-to-end, but object-oriented methodologies have nothing comparable. Because conventional methods stress functionality over object partitioning, showing the operation of the overall system via functions is consistent with the methodology approach in general. In contrast, object-orientation concentrates on objects as stand-alone reusable components instead of how those components tie together. Object-oriented methodologies partition objects and their relationships into several models which represent different views of the objects and their interactions. These views are generally presented in the form of static architectures and dynamic behavior. There are typically multiple instances of each view, with each instance representing a fragment of the system. The observer must deal not only with these multiple views of the system, but also with fragments of the system at a time. Much effort is required to obtain a synergistic understanding of the system being modeled as a whole.

One of the key mechanisms for ensuring system completeness is to trace through threads of control. A thread of control is a path through a sequence of operations representing a particular scenario in the system being modeled. Threads of control integrate the overall flow of data, control, events, and timing up to the system level. They provide a means by which the system may be analyzed and understood as a whole. Thread of control information is desirable in two ways: first, it ensures that all of the pertinent objects exist to support the system as a whole; and secondly, if timing is critical in the system, tracing through the threads of control may identify essential timing constraints and potential bottlenecks. The presence of multiple threads of control is an indicator that the system will need to be partitioned into processes, that is, separate executable entities. Identification of these critical areas early on will drive decisions concerning process allocation, and how data will be transferred, accessed, and shared. Because of their significance, successfully representing threads of control for a system being modeled greatly enhances understanding the operation of the system as a whole.

Viewing a system in terms of the processes which make it up adds additional complications. On the one hand, large-scale real-time distributed systems reconcile competing demands for resources by partitioning the system into multiple processes. On the other hand, object-oriented technology strives to partition a system by objects where all data and operations associated with an object are encapsulated within the object. The partitionings for processes and objects appear to be orthogonal in this context when threads of control are considered. Hence, the partitioning goals associated with object-oriented and distributed systems are conflicting.

This paper introduces a means of representing threads of control and their associated classes/objects to better illustrate how the system operates. Towards this end, an analysis of five predominant object-oriented analysis and design methodologies was performed.

The methodologies reviewed include Coad and Yourdon [Coad91a] [Coad91b], Shlaer and Mellor [Shlaer88] [Shlaer91], Booch [Booch], Firesmith [Firesmith], and Rumbaugh [Rumbaugh]. While some of the methodologies reviewed describe both analysis and design (Coad and Yourdon, Firesmith and Rumbaugh), Shlaer and Mellor focus more on analysis and Booch on design. The distinction between object-oriented analysis and design is not precise. There are inconsistencies in the research about what comprises each, and the lines between analysis and design in object-orientation are blurred [Berard], [Korson]. It is not the goal of this paper to distinguish between object-oriented analysis and design. Instead, the intent is to focus on the constructs necessary to support end-to-end processing during object-oriented analysis and design as opposed to object-oriented programming.

A review of methodologies indicates that both static (class/object architecture) and dynamic (control and data flow) representations of systems exist; however, threads of control are only minimally represented and are fragmented. This paper extends the static and dynamic concepts by introducing a representation which overlays dynamic flow (via thread of control information) onto a static structure. In order to combine dynamic and static representations to show end-to-end processing, class/objects are grouped so that they may be represented at a higher level of abstraction. Determining how the class/objects were to be grouped resulted in a partitioning problem. To simplify the partitioning problem, the proposed grouping approach is performed in two phases. The first phase involves a logical grouping of class/objects. The logical groupings are further refined with thread of control (state, control, and data flow) information, providing a coarse-grained partitioning referred to as *logical composite classes*. The second phase further extends the partitioning using process partitioning criteria based on other thread of control information involving communication and timing constraints to develop *process composite classes*. The introduction of these design-level constructs provides the basis for a mechanism to automate different instances of composite process classes for timing and concurrency comparisons.

ANALYSIS OF CURRENT OBJECT-ORIENTED ANALYSIS AND DESIGN METHODOLOGIES

Each of the five methodologies reviewed provided some means of representing both the static architecture and dynamic behavior of a system. The following is a brief description of the techniques each methodology employs for representing static and dynamic views of a system and an overall assessment of these techniques.

Static Architecture

The static architecture refers to a non-temporal representation of the system. A static representation of the system is generally reflected by some variation of entity-relationship diagram. Entities, in this context, are either classes or objects. The distinction between classes and objects is that a *class* serves as a template for defining the characteristics of an object. An *object* is a software abstraction that models a concept, abstraction, or thing

which represents the application domain (analysis) or the solution space (design). To further distinguish the two, an object is an instance of a class. Further, a *concrete class* is a class for which object instances may be created, as opposed to an *abstract class* for which objects may not exist.

The static models and diagrams associated with the methodologies reviewed are summarized in Figure 1. The diagrams for each methodology which depict classes/objects and their relationships are those listed first in the Class/Object Representation row of Figure 1. These diagrams, when used in a general context, will be termed class/object diagrams since they generally contain more information than what is usually associated with an entity-relationship diagram as is described below.

| | Coad and Yourdon | Shlaer and Mellor | Booch | Firesmith | Rumbaugh |
|--|---|--|---|---|---|
| Class/ Object Model(s) | <ul style="list-style-type: none"> • Class-&-Object Layer • Structure Layer • Attribute Layer • Service Layer | <ul style="list-style-type: none"> • Information Model | <ul style="list-style-type: none"> • Class Structure • Object Structure | <ul style="list-style-type: none"> • Class Model • Object Model | <ul style="list-style-type: none"> • Object Model |
| Class/ Object Representation(s) | <ul style="list-style-type: none"> • Class-&-Object Diagram • Gen-Spec Structure • Whole-Part Structure | <ul style="list-style-type: none"> • Information Structure Diagram • Inheritance Diagram | <ul style="list-style-type: none"> • Class Diagram • Object Diagram • Class Template | <ul style="list-style-type: none"> • General Semantic Net • Interaction Diagram • Classification Diagram • Composition Diagram • Class Specification | <ul style="list-style-type: none"> • Object Diagram • Generalization Notation • Aggregation Notation |
| <p>The above representations all include classes/objects, relationships, and attributes. All include operation specifications with the exception of Shlaer and Mellor.</p> | | | | | |

Figure 1. Static Models and Diagrams

An object, or the class template for the object, is usually defined in terms of its attributes and operations. *Attributes* are fields which describe data values within a class/object, and *operations* are functions performed by a class/object. Two methodologies, Coad and Yourdon and Firesmith, represent attributes and operation specifications on their Class-&-Object Diagram and Object Diagram respectively. Shlaer and Mellor include only attributes on their Information Structure Diagram. Both Booch and Firesmith use a separate means for representing attributes and operation specifications. Booch describes a Class Template, and Firesmith a Class Specification.

Relationships in a class/object diagram refer to associations between two or more classes/objects indicating some type of structural or semantic link. In addition to simple

association, two special types of relationships exist in most object-oriented methodologies: *is-a* and *has-a* relationships.

Is-a relationships introduce the concepts of generalization, specialization, and inheritance. A *generalization* is a higher level of abstraction of a class. For example, the class *animal* is a generalization of the classes *cat* and *dog*. Because it is a generalization, animal is a *superclass* of cat and dog. Animal might be described as furry and four-legged. While both the cat and dog are furry and four-legged, the cat meows and the dog barks. Because they are *specializations* of animal, cat and dog are *subclasses* of animal; they *inherit* the characteristics of being furry and four-legged, but they extend the animal class by adding special characteristics such as meowing or barking. Specifically, a subclass inherits the attributes and operations of its superclass, and extends it further with additional attributes or operations. Several of the methodologies contained special diagrams to represent the *is-a* relationship: Coad and Yourdon/Gen-Spec Structure, Shlaer and Mellor/Inheritance Diagram, Firesmith/Classification Diagram, and Rumbaugh/Generalization Notation.

Has-a relationships depict an *aggregation* of class/objects. A class/object which contains at least one other class/object is referred to in this paper as a *composite class/object*. For example, the composite class/object *car* is made up of doors, wheels, an engine, etc. Conversely, a class/object which does not contain other classes/objects is termed an *atomic class/object*. Several methodologies represented aggregation associations with special notation: Coad and Yourdon/Whole-Part Structure, Firesmith/Composition Diagrams, and Rumbaugh/Aggregation Notation.

Dynamic Behavior

Dynamic behavior is behavior attributable to timing and the flow of information in the system being modeled. The information typically represented in dynamic models includes state, control and data flow, and timing information. The *states*, or modes, of a class/object reflect the attribute values of a class/object at a given point in time. *State information* contains the states of a specific class/object, and the operations or events that effect transitions between the class/object's states. State transition diagrams (STDs) are the most common representation of state information. STDs are generated for each class/object which has interesting behavior. All of the STDs in the methodologies reviewed contained states, events, transitions and operations with the exception of Coad and Yourdon's Object State Diagram which contained only states and transitions. *Control flow information* describes the control and sequencing of a message within or between classes/objects. It is most often represented in a control flow diagram (CFD). A *message* may be a request for service, event, or passing of data. *Data flow information* describes the flow of data among classes/objects via their operations. A data flow diagram (DFD) is commonly used to show data flow among the class/objects. *Timing information* contains the duration of operations within and between class/objects and is usually associated with control flow information.

Control flow information is only minimally represented on the Coad and Yourdon Class-&-Object diagram via arrows between the class/objects which represent message connections. Booch shows control flow only in the context of a Timing Diagram which displays objects and the invocations of their operations along a time axis. Rumbaugh's primary mechanism for control flow is his State Diagram; although control may also be shown on a DFD but is considered redundant. Both Shlaer and Mellor and Firesmith combine control and data flow information onto one diagram, the Action DFD and Object-Oriented CFD respectively. Neither Coad and Yourdon or Booch describe representations for data flow. For timing information, only Booch (as previously mentioned) and Firesmith provide a timing diagram. Coad and Yourdon allow that a time requirement may be annotated with the specification of a particular class/object. Shlaer and Mellor describe time only in the context of threads of control which is addressed later in this paper.

The dynamic models and diagrams associated with the methodologies reviewed are summarized in Figure 2. The first bullet in each cell lists the model used to address each type of dynamic information, and the second bullet lists the diagram (or diagrams) the methodologies use to represent information.

| | Coad and Yourdon | Shlaer and Mellor | Booch | Firesmith | Rumbaugh |
|---|---|--|--|--|--|
| State Information | <ul style="list-style-type: none"> • Services Layer • Object State Diagram | <ul style="list-style-type: none"> • State Model • STD | <ul style="list-style-type: none"> • Class Structure • STD | <ul style="list-style-type: none"> • State Model • STD | <ul style="list-style-type: none"> • Dynamic Model • State Diagram |
| Control Flow Information | <ul style="list-style-type: none"> • Services Layer • Message Connections on Class-&-Object Diagram | <ul style="list-style-type: none"> • Process* Model • Action DFD | <ul style="list-style-type: none"> • Object Structure • Timing Diagram | <ul style="list-style-type: none"> • Control Model • Object-Oriented CFD | <ul style="list-style-type: none"> • Dynamic/Functional Models • State Diagram/DFD |
| Data Flow Information | <ul style="list-style-type: none"> • N/A | <ul style="list-style-type: none"> • Process* Model • Action DFD | <ul style="list-style-type: none"> • N/A | <ul style="list-style-type: none"> • Control Model • Object-Oriented CFD | <ul style="list-style-type: none"> • Functional Model • DFD |
| Timing Information | <ul style="list-style-type: none"> • Services Layer • Timing Textual Annotation in Class-&-Object Specification | <ul style="list-style-type: none"> • N/A | <ul style="list-style-type: none"> • Object Structure • Timing Diagram | <ul style="list-style-type: none"> • Timing Model • Timing Diagram | <ul style="list-style-type: none"> • N/A |
| STD: State Transition Diagram, DFD: Data Flow Diagram, CFD: Control Flow Diagram | | | | | |
| <i>*Process is a transform in this context.</i> | | | | | |

Figure 2. Dynamic Models and Diagrams

Dynamic behavior is the basis of information upon which threads of control are built; however, dynamic behavior models only fragments of the system. State information is associated with a particular class/object. Control and data flows are usually represented between a particular group of classes/objects. Timing diagrams depict time durations of

operations associated with segments of the system. Threads of control track the information provided in the dynamic behavior models along a particular path which is representative of a system scenario. In that threads of control integrate the puzzle pieces which make up the system, their representation is fundamental to understanding how a system operates as a whole.

THREADS OF CONTROL

A thread of control is a path which traces a sequence of operations among or within objects or classes. This path represents a scenario which may be used during analysis, design, or testing to trace through the model. Threads of control are valuable for analyzing the model for completeness to ensure that all aspects of the system being modeled are represented. Additionally, for real-time systems, they are essential in identifying real-time processing requirements for timing constraints and bottlenecks. Threads of control represent the integration, along a particular path, of the state, control flow, data flow, and timing data contained in the dynamic behavior models. State information is needed because the thread of control may vary depending on the state of the class/object. Timing, data flow and control flow data provide the sequencing information, data required along a particular sequence, and associated duration.

Coad and Yourdon presented thread of control information in a cursory fashion via message connections on their Class-&-Object diagram; although descriptive information about the threads of control as related to a particular class/object may be contained within that class/object's specification. In all of the methodologies reviewed, only Shlaer and Mellor had a clear representation of the relation between the states of a class/object and the threads of control associated with the states using a thread of control chart; however, the tie back to the associated class/object was not apparent and none of the data associated with the flow was represented. The Timing Diagram was the only mechanism available in Booch's methodology which reflected thread of control related information. While it tied operations, sequencing, and times to objects, it was deficient in representing data and state information. Firesmith provided three different diagrams containing various thread of control information: an object-oriented control flow diagram for each major thread of control, a thread-level interaction diagram to show the interactions of classes/objects for a given scenario, and a timing diagram for each thread of control. Rumbaugh used event trace diagrams to show the sequencing of events in a system; however the information provided in this diagram depicted only the event sequencing and the class/object impacted by the event. Thread of control representations associated with the methodologies reviewed are summarized in Figure 3.

The review of methodologies for thread of control information indicated that none of the methodologies covered all of the information associated with threads of control. Firesmith's method appeared to provide the best and most comprehensive thread of control information of the methodologies reviewed, but the information is spread over several diagrams and is therefore difficult to assimilate.

| | Coad and Yourdon | Shlaer and Mellor | Booch | Firesmith | Rumbaugh |
|---------------------------------|--|--|---|---|---|
| Thread of Control (TOC) | Message Connections on Class-&Object Diagram. TOC in Class-&Object Specification in bullet list format or in Service Chart. | Thread of Control Chart shows events and states occurring in a thread and associated times. | Timing Diagram shows objects and operations sequence and duration. | TOC Object-Oriented CFD, Thread-level Interaction Diagrams, TOC Timing Diagram | Event traces - shows event sequencing and the associated class/objects |
| State Information | no | yes | no | no | no |
| Control Flow Information | yes | yes | yes | yes | yes |
| Data Flow Information | no | no | no | yes | yes |
| Timing Information | yes | yes | yes | yes | no |
| Associated Class/Object | yes | no | yes | yes | yes |

Figure 3. Thread of Control Representations

Because of the significant role that threads of control play in understanding the overall operation of a system, an effective means of representing them in object-oriented analysis and design is needed. Current methodologies tend to fragment this information showing only segments of the system at a time, using multiple models for different views of these segments. The approach to integrating threads of control in object-oriented analysis and design described here begins by abstracting classes/objects at a higher level. The rationale for this higher level of abstraction is two-fold: first, end-to-end processing is easier to show, as well as understand, at a higher level of abstraction; second, thread of control information is attached at the higher level of abstraction which lessens the amount and complexity of the information to be handled. Abstracting classes/objects at a higher level implies that classes/objects are aggregated into larger groups based on some criteria, such as being logically related to each other. These logically related groups make up a logical view of the system.

LOGICAL COMPOSITE CLASSES

A *logical view* represents the groupings of classes/objects which are logically related into higher levels of composition. Partitioning into groups is usually based on engineering judgment, and minimizing the associations, aggregations and generalizations between groups. Traditionally, the rationale for grouping classes/objects is for partitioning large projects, and to provide a means of understanding the overall system and its interfaces. A logical view which overlays a class/object diagram might look something like what is shown in Figure 4.

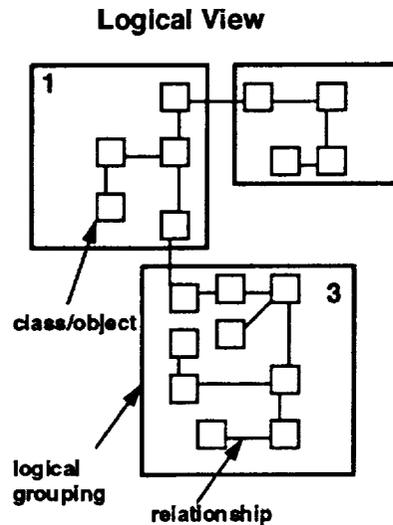


Figure 4. Groupings for a Logical View

Although no formal method for grouping classes/objects was presented in the methodologies reviewed, each methodology touched on the concept in some fashion. Yet, the terminology and approaches associated with the groupings of classes/objects varies among the different methodologies. Coad and Yourdon use the term Subject. Subjects are initially created by identifying the uppermost class, the parent, in each is-a or has-a structure and calling it a subject. Subjects are further refined by minimizing the relationships and message connections between the subjects. Shlaer and Mellor use a top-down approach to grouping. They begin by identifying the different domains which make up a system and partition large domains into subsystems. It is for each of these subsystems that the class/object diagrams are constructed. Booch combines classes/objects into modules which are actually physical representations in that they are intended to represent software modules. Modules may be further logically grouped into subsystems. Firesmith groups classes/objects into subassemblies which ultimately make up an assembly. He describes several approaches to identifying subassemblies depending on the situation involved. Some of the approaches involve bottom-up development where class/objects are identified and then grouped based on the criteria dictated by the approach, such as coupling and cohesion criteria. Other approaches, such as recursion, begin with top-down development by identifying parent subassemblies, and recursively defining other subassemblies as needed. Rumbaugh introduces modules which are logical groupings of the class/objects and associated relations defined in his object model. Modules are the lowest level subsystems. The terminology, models and representations associated with each of the methodologies for a logical view are shown in Figure 5.

| | Coad and Yourdon | Shlaer and Mellor | Booch | Firesmith | Rumbaugh |
|-----------------------|------------------------------------|---|---|---|---|
| Terminology | Subjects | <ul style="list-style-type: none"> • Domain • Subsystem | <ul style="list-style-type: none"> • Subsystem • Module | <ul style="list-style-type: none"> • Assembly • Subassembly | <ul style="list-style-type: none"> • Subsystem • Module |
| Model | Subject Layer | ** <ul style="list-style-type: none"> • Domain Chart • Subsystem Relationship Model • Subsystem Communication Model • Subsystem Access Model | Module Architecture | Assembly Model | N/A |
| Representation | Class-&-Objects Diagram | | <ul style="list-style-type: none"> • Subsystem Diagram • Module Diagram | <ul style="list-style-type: none"> • Context Diagram • Assembly Diagram | Object Diagram (modules only) |

** No distinction between model and representation

Figure 5. Logical View Representations

While these methodologies all describe logical groupings, they do not use this construct in conjunction with thread of control information to represent end-to-end processing. It is in this context that the logical composite class construct is introduced. A *logical composite class* is a grouping of classes/objects which are logically related and further refined/extended by integrating thread of control information. The rationale for logical composite classes is that they provide a mechanism for representing end-to-end threads of control through class/object groupings combining both static architecture and dynamic behavior. They are also a precursor to process composite classes which further refine groupings using process partitioning criteria. Process composite classes are detailed in a later section. These constructs should be viewed as design-level classes which can be integrated into a design language. Instances of this class are the actual groupings and their associated data.

The methodology used to generate logical composite classes is a bottom-up approach which begins with the initial groupings formed from the logical view. Next, the pertinent state, data and control flow information required for threads of control is aggregated for each logical grouping. As previously described in the dynamic behavior models, this information is already available in fragmented form at the class/object level. To aggregate the information means to recompose the information at the class/object level to the level of abstraction of the logical groupings in a summarized form. This aggregated information is assessed at the boundaries of the logical groupings by focusing on the information required between the *boundary* classes/objects. The boundary class/objects are those class/objects in the logical view that play an interface role between the groups defined in the logical view. The aggregated information is attached to the associated logical grouping. Groupings are then refined to minimize connections among groups. Figure 6

shows the logical composite classes which evolved from the logical view groupings of Figure 4.

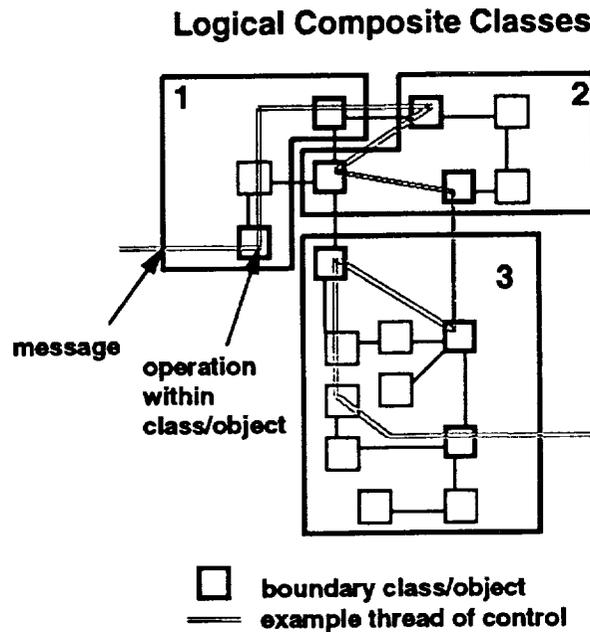


Figure 6. Logical Composite Class Representation

The philosophy of using composite classes as aggregations of class/objects is probably most closely associated with how Coad and Yourdon identify and refine subjects, since subjects evolve partially out of has-a relationships. Refining the groupings as development continues is consistent with Firesmith's recursive approach to development. However, the logical composite class extends these concepts further by introducing a design-level construct which contains a grouping of classes/objects at a higher level of abstraction and attaches aggregated data representing thread of control information to those constructs.

While all of the methodologies described logical groupings, none of them addressed the refinement of these groupings for processes as is required in real-time and distributed systems. In this paper, the concept of grouping is extended even further using the process composite class construct as a mechanism for refining groupings along process lines.

PROCESS COMPOSITE CLASS

A *process view* represents the mapping of class/objects to *processes*. In this context, processes are entities implemented in software that may execute concurrently and compete for resources. The introduction of multiple threads of control necessitates partitioning

systems into processes. The logical composite classes shown earlier might contain multiple threads of control as shown in Figure 7.

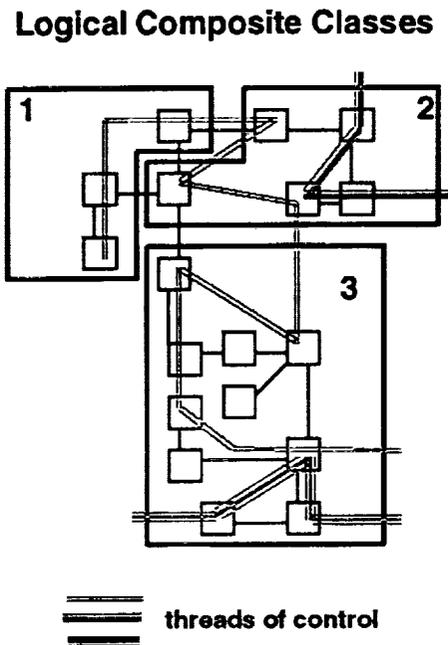


Figure 7. Logical Composite Classes with Multiple Threads of Control

Of the methodologies reviewed, the only methodology that provided a model for processes was the Process Architecture model presented in Booch. This model described templates for processes and processors. These templates contained information concerning the characteristics of the computer, processes associated with each processor, priority for each process and the scheduling approach. However, the model lacked any transition or correlation to Booch's previously described class structure, object structure or module architecture models. Additionally, no criteria for how processes should be allocated or identified was provided.

The need for process partitioning has long been recognized in the real-time development community. The merging of this technology with object-orientation is still in its infancy. The key criteria for process partitioning have to do with communication and timing. In terms of communication, the ideal is to minimize communication between processes by grouping classes/objects which interface extensively within a process, thereby reducing the interaction between groups. The interface between groups is referred to as *coupling*, and within a group, *cohesion*. An excellent discussion on the coupling and cohesion of objects and modules is presented in [Berard].

Timing criteria affect process partitioning in a number of ways. For example, those classes/objects whose operations support services which must be performed within a specified time should be grouped in an independent process. Classes/objects whose operations support services which perform on different cycles, sporadically, or at a low level of priority should be separated into different processes. As previously mentioned, threads of control may be used to trace through critical paths in a system to determine total execution criteria. While a determination may be made to add processes due to timing constraints, the tradeoff between adding these processes versus the overhead to run them must be weighed. Additionally, the more processes that are added, the more complex the system becomes. A representative listing of partitioning criteria for processes is provided in [Nielsen].

The construct introduced in this paper to represent the partitioning of systems into processes, is the process composite class. A *process composite class* is a grouping of classes/objects originating from the logical composite class groupings and further refined based on process partitioning criteria. The logical composite classes already represent an initial partitioning based on the existence of interactions between groups. The methodology for developing process composite classes begins by extending these logical composite classes with timing information. The timing information associated with each logical composite class is assessed. Class/objects or class/object groupings which have distinguishing timing criteria such as being time critical or the other extreme, low priority, are extracted from within the logical composite classes. Weights may then be assigned to interfaces between modified groupings as a function of the number of data/control flows among the groupings. These weights determine the need for further repartitioning based on changed interactions between groups resulting from the previous repartitioning based on timing. Weights reflect the magnitude of communications between the groups. Repartitioning is performed as needed to achieve total execution time criteria. Figure 8 highlights how these sequences of repartitionings might look. Beginning with the grouping of the logical composite classes from Figure 7, Figure 8 shows subsequent groupings into process composite classes based on various process partitioning criteria. Keeping track of these numerous classes/objects, the interrelationships among them, the threads of control through them, and the partitioning criteria needed to determine the potential groupings into composite structures, quickly becomes a complex problem which is well suited for a database environment.

The formulation of groupings into process composite classes involves taking the thread of control information attached to the logical composite class, and applying process partitioning criteria with system constraints to result in process composite classes. Classes and their associated attributes, operations and state data are contained in a database. The relationships that tie operations to particular state values or changes in attribute values are also maintained. In the context of a logical composite class, thread of control information is extracted from the appropriate classes. That is, the class/objects whose operations are invoked along that thread of control, and the attributes and data impacted or used in conjunction with those operations, are linked to the thread of control. Additional information associated with the particular thread of control such as operation

precedence, identification of time critical operations (priorities and deadlines), priority and timing constraints, and communication interface requirements is also included.

Process Composite Classes

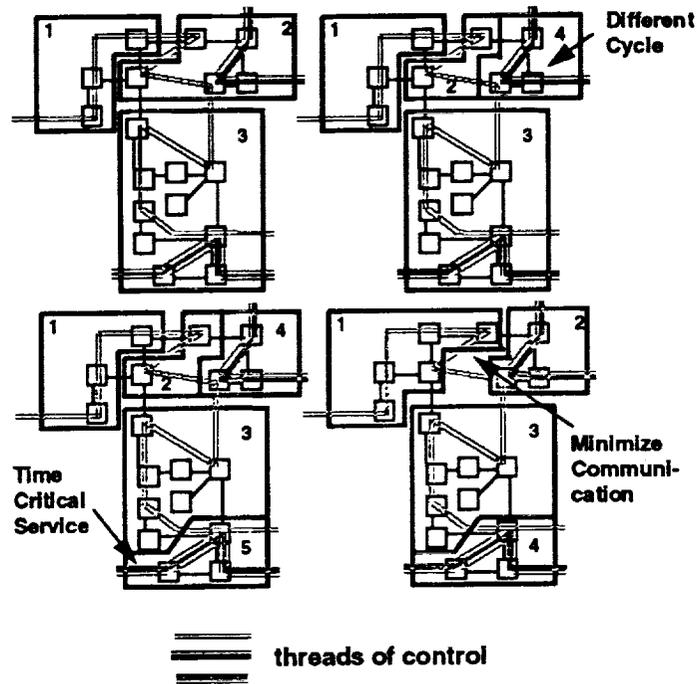


Figure 8. Repartitionings of Process Composite Classes

After threads of control are enumerated, interrelationships may be identified and assessed. For example, different threads of control may use different operations within a class. Interrelationships may be involved if one thread of control alters attribute values by invoking a particular operation in a class where these attribute values are also used by another operation invoked by a separate thread of control. The intra-dependencies of attributes affected by operations within a given class is maintained in the database. These intra-dependencies must be considered among the various threads of control. The interdependencies along various threads of control between logical composite class groupings must also be considered. These dependencies and their magnitude provide much of the data needed to make process partitioning decisions.

The process composite class definition can be augmented by algorithms which provide optimal solutions to allocations. Given the proper criteria, these algorithms can provide solutions using various methods such as graph-theoretic allocation or a heuristic branch and bound allocation that minimizes or maximizes performance objectives

[Horowitz], [Reeves]. Typical constraints minimize the cost of running the total system by partitioning the process composite classes efficiently. The partitioning resulting from the process partitioning criteria, combined with system constraints such as communication bandwidth, processor speed or concurrency limitations, provide the information needed to define the performance objectives. Figure 9 depicts the overall formulation of groupings into process composite classes.

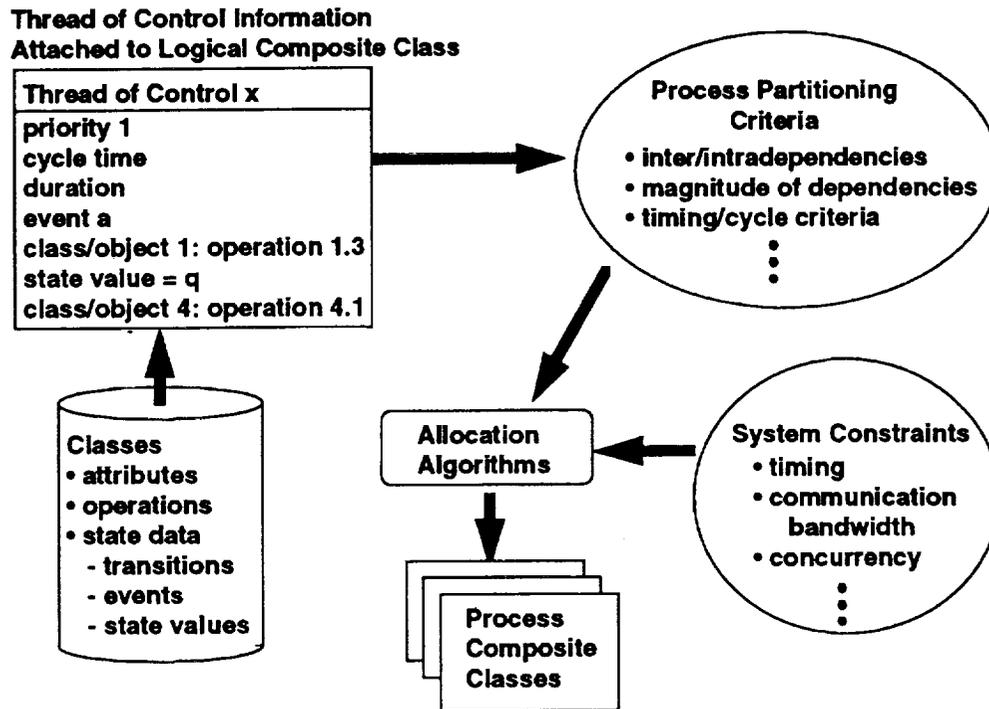


Figure 9. Formulating Process Composite Classes

The results of this work are being used to develop a streamlined methodology for use with distributed, real-time applications. Basic class/object static architectures and dynamic behaviors will be drawn from the strengths of the methodologies reviewed and consolidated. The logical and process composite class structures will provide a layer above these other constructs and will be integrated in a design language. Integrating this concept into a design language provides a means of representing the structure graphically, building a database, and generating consistency checks. Additionally, it provides a basis for an automated mechanism so that regrouping for logical and composite class structures, and the application of algorithms to these structures, may be easily accomplished for efficiency comparison purposes.

FUTURE WORK

Several issues have arisen as a result of this research which require further investigation. These issues focus on specific cases where object-oriented and distributed system partitionings are in conflict. One case concerns the fact that distributed systems sometimes require that parts of the same object be in multiple locations. For example, different operations may be required on the same object depending on where it is located in the system. This requirement is contrary to all of the attributes and operations associated with an object being encapsulated within the object. Another case is one in which the various operations contained in an object may have different timing constraints. For example, one operation may be along a time critical thread of control while another may not. The first inclination would be to group the object into a process in accordance with the highest priority operation. The down side of this, however, is that all of the other information related with that object, such as the secondary operations, and threads of control and objects associated with those secondary operations, are then grouped into the same time critical process. These cases and others like them require further exploration in order to integrate solutions into the process partitioning approach.

SUMMARY

The results of this research indicate that current object-oriented analysis and design methodologies' representations do not provide a clear understanding of the end-to-end processing which defines system operation. This research has introduced logical and process composite classes that act as structures for representing groupings of class/objects. These structures reflect classes/objects and the threads of control through those classes/objects. Further study is needed to extend these structures into a design language, and refine the partitioning conflicts which arise between objects and processes.

REFERENCES

- [Berard] Berard, Edward V., *Essays on Object-Oriented Software Engineering, Volume I*, Prentice Hall, Englewood Cliffs, NJ, 1993.
- [Booch] Booch, Grady, *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing, Redwood City, CA, 1991.
- [Coad91a] Coad, Peter and Yourdon, Edward, *Object-Oriented Analysis, Second Edition*, Yourdon Press, Englewood Cliffs, NJ, 1991.
- [Coad91b] Coad, Peter and Yourdon, Edward, *Object-Oriented Design*, Yourdon Press, Englewood Cliffs, NJ, 1991.
- [Fichman] Fichman, Robert G., and Kemerer, Chris, F., "Object-Oriented Analysis and Design Methodologies Comparison and Critique," *Computer*, Vol. 25, No. 10, October 1992, pp. 22-39.
- [Firesmith] Firesmith, Donald G., *Object-Oriented Requirements Analysis and Logical Design - A Software Engineering Approach*, John Wiley & Sons, New York, NY, 1993.

- [Horowitz] Horowitz, E., and Sahni, S., *Fundamentals of Computer Algorithms*, Computer Science Press, Inc., Rockville, Maryland, 1978.
- [Korson] Korson, T., and McGregor, J. D., "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol. 33, No. 9, September 1990, pp. 40-60.
- [Nielsen] Nielsen, Kjell, *Object-Oriented Design with Ada*, Bantam Books, New York, New York, 1992.
- [Reeves] Reeves, Colin R., *Modern Heuristic Techniques for Combinatorial Problems*, John Wiley & Sons, Inc., New York, NY, 1993.
- [Rumbaugh] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenzen, W., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [Shlaer88] Shlaer, Sally, and Mellor, Stephen J., *Object-Oriented Systems Analysis - Modeling the World in Data*, Yourdon Press, Englewood Cliffs, NJ, 1988.
- [Shlaer91] Shlaer, Sally, and Mellor, Stephen J., *Object Lifecycles - Modeling the World in States*, Yourdon Press, Englewood Cliffs, NJ, 1992.

Integrating End-to-End Threads of Control into Object-Oriented Analysis and Design

Eighteenth Annual Software Engineering Workshop
NASA/Goddard Space Flight Center
December 1-2, 1993

Janet E. McCandlish
TRW System Development Division
Huntsville Operations

Dr. James R. MacDonald
ALPHATECH, Inc.

Dr. Sara J. Graves
University of Alabama
in Huntsville

 System Development Division
Huntsville Operations

1

Problems

- **Current object-oriented analysis and design methodologies fall short in their representation of end-to-end processing**
 - system is represented with multiple views
 - only pieces of the system are represented
 - people have difficulty in seeing how system operates
- **Goals associated with object-oriented and distributed systems are conflicting**
 - large-scale real-time distributed systems reconcile competing demands for resources by partitioning the system into multiple processes
 - object-oriented technology strives to partition a system by objects where all data and operations associated with an object are encapsulated within the object
 - the partitionings for processes and objects appear to be orthogonal in this context when threads of control are considered

 System Development Division
Huntsville Operations

2

Solution/Approach Overview

- **Represent threads of control and their associated class/objects to better illustrate how the system operates**
 - Five current object-oriented analysis and design methodologies assessed: Coad and Yourdon, Shlaer and Mellor, Booch, Firesmith, and Rumbaugh
 - Introduce a representation which overlays dynamic flow (threads of control) onto a static structure
- **Group class/objects at higher level of abstraction for process partitioning**
 - Combining dynamic and static representation to show end-to-end processes requires some grouping of classes/objects at higher levels
 - To simplify partitioning problem, grouping is two-phased: (1) logical groupings, further refined with thread of control information (provides a coarse-grained partitioning) (2) process groupings, extend logical groupings with process partitioning criteria

Background

- **Static Architecture**
 - non-temporal representation of the system
 - typically depicted as an enhanced entity-relationship diagram
- **Dynamic Behavior**
 - behavior attributable to timing and flow of information
 - may include state, control flow, data flow and timing information
- **Thread of Control**
 - path which traces a sequence of operations among or within objects or classes
 - represents a scenario which may be used during analysis, design, or testing to trace through the model for completeness and real-time processing requirements
- **Static and dynamic representations exist, but thread of control representation is limited**

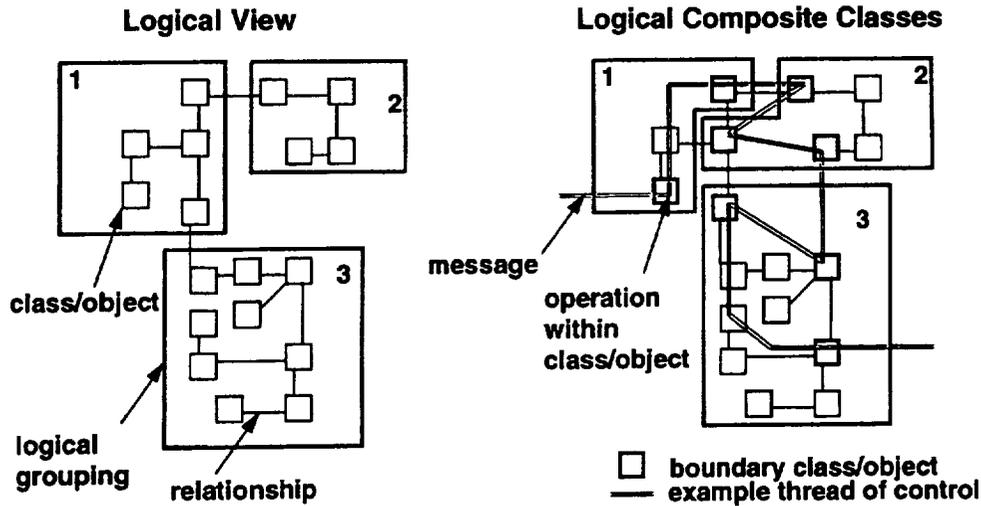
Thread of Control Representations

| | Coad and Yourdon | Shlaer and Mellor | Booch | Firesmith | Rumbaugh |
|--------------------------|---|---|--|--|--|
| Thread of Control (TOC) | Message Connections on Class-&Object Diagram. TOC in Class-&Object Specification in bullet list format or in Service Chart. | Thread of Control Chart shows events and states occurring in a thread and associated times. | Timing Diagram shows objects and operations sequence and duration. | TOC Object-Oriented CFD, Thread-level Interaction Diagrams, TOC Timing Diagram | Event traces - shows event sequencing and the associated class/objects |
| State Information | no | yes | no | no | no |
| Control Flow Information | yes | yes | yes | yes | yes |
| Data Flow Information | no | no | no | yes | yes |
| Timing Information | yes | yes | yes | yes | no |
| Associated Class/Object | yes | no | yes | yes | yes |

Logical View

- The *Logical View* represents groupings of classes/objects which are logically related. Partitioning into groups is based on:
 - engineering judgement
 - minimizing the associations, aggregations and generalizations between groups
- **Rationale for Logical Groupings**
 - Partitioning for large projects
 - Means of understanding overall system and interfaces
- **Terminology**
 - Subjects - *Coad and Yourdon*
 - Domains/Subsystems - *Shlaer and Mellor*
 - Subsystems/Modules - *Booch, Rumbaugh*
 - Assemblies/Subassemblies - *Firesmith*

Logical Composite Class Representation



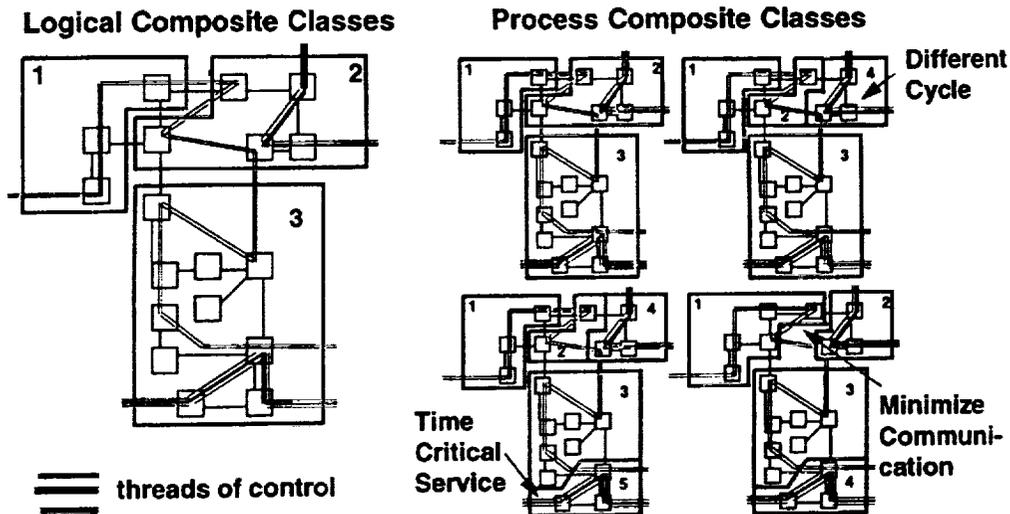
Process View

- The *Process View* represents the mapping of class/objects to *processes* (entities implemented in software that may execute concurrently and compete for resources).
- The introduction of multiple threads of control is a major reason for partitioning the systems into processes.
- The only methodology reviewed that provides a model for processes is Booch's Process Architecture.
 - Describes templates for processes and processors
 - Provides no transition from his other models (class structure, object structure or module architecture) to the process architecture

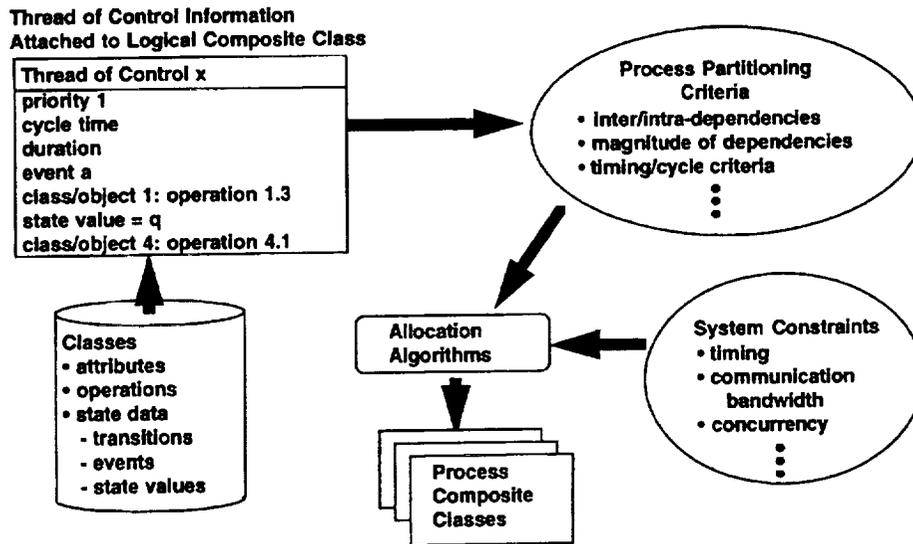
Process Composite Class

- A *process composite class* is a grouping of classes/objects originating from the logical composite class groupings and further refined based on process partitioning criteria
- **Process Partitioning Criteria**
 - Communication
 - » minimize communication between processes
 - Timing
 - » class/objects whose operations support services which must be performed within a specified time
 - » class/objects whose operations support services which perform on different cycles, sporadically, or at a low level
 - » adjust process groupings as needed to meet total execution time criteria

Process Composite Class Representation



Formulating Process Composite Classes



Ongoing Work

- **Represent the logical and process composite class structure in a design language**
 - provides a means of representing the structure graphically, building a database, and generating consistency checks
 - basis for automated mechanism so that regrouping for logical and composite class structures may be easily accomplished for efficiency comparison purposes
- **Address specific cases where object-oriented and distributed system partitionings are in conflict and integrate solutions into process partitioning approach. Examples include:**
 - distributed systems sometimes require that parts of the same object be in multiple locations
 - one operation in an object may be along a time critical thread of control while another may not; requires that entire object and all associated threads and objects be grouped into time-critical process

Summary

- **Current object-oriented analysis and design methodologies representations do not provide the viewer with a clear understanding of the end-to-end processing which defines system operation**
- **This research has introduced logical and process composite classes that act as structures for representing groupings of class/objects. These structures reflect classes/objects and the threads of control through those classes/objects.**
- **Further study is needed to:**
 - extend these structures into a design language
 - refine the partitioning conflicts which arise between objects and processes