OBJECT-ORIENTED ANALYSIS AND SPECIFICATION FOR REAL-TIME SYSTEMS

by

EUNICE YOON-GIL MOON, B.S.

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

December, 1990

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to Dr. Bagert for his suggestions and guidance in the preparation of this thesis. I also wish to express my appreciation to Dr. Marcy for his helpful comments while serving on my committee.

I also want to thank my husband, Nathan, for his encouragement and support. Thanks extend to my parents in California for their understanding and love.

CONTENTS

ABSTRACT

It is important to provide a complete, consistent, and feasible requirements specification, especially for complex real-time systems. To maximize the effectiveness of object-oriented software development, it is ideal to apply the object-oriented approach to every stage of the software life cycle. In this research, an object-oriented analysis methodology which can be applied to real-time systems (OOART) has been developed and applied to an existing system. A message passing technique which employs the priority queue, and embeds the time constraints in each message, has also been developed. To represent the OOART in the resulting requirements specification, a set of graphical notations which not only supports the concepts of the object-oriented approach, but also supports the characteristics of real-time systems, has been utilized. As a case study, the OOART has been applied to a single wafer cleaning system which implements a distributed real-time control of semiconductor process equipment.

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Abbott defines requirements as "any function, constraint, or other property that must be provided, met, or satisfied to fill the needs of the system's intended user(s)" [25]. Requirements analysis is the process of determining requirements for a system. Therefore, requirements analysis means the process of defining what the system must do, not how the system will be implemented. The usual outcome of analysis is a document called the Software Requirements Specification (SRS) which describes the expected external behavior of the software system to be built. To avoid confusion at the later stages of a software life cycle, it is important to provide a complete, consistent and feasible requirements specification, especially for large and complex systems [27].

A variety of analysis methods, requirements specification techniques and their underlying models have been developed. No matter what method is used, the resulting requirement specification should be able to serve as the basis for the subsequent phase of software development: software design. The object-oriented design

methodology has become popular recently [8]. Booch defines object-oriented development as an approach to software design in which the decomposition of a system is based upon the concept of an object [4]. Object-oriented design methodology is fundamentally different from traditional functional methods, where the basis for decomposition is that each module in the system represents a major step in the overall process [4]. An object-oriented design should then be used to create optimal object-oriented implementations [6]. Similarly, object-oriented methods should be applied to specification when the use of object-oriented design is foreseen [7]. In other words, to maximize the effectiveness of object-oriented software development, it is ideal to apply the object-oriented approach to every stage of the software life cycle. However, as a general object-oriented software development methodology, many researchers used structured analysis for developing the specification, and then make a transition from a structured specification to an object-oriented design [3, 4, 7]. Until recently, few techniques had been developed for object-oriented analysis and requirements specification. Balin described an object-oriented requirements specification technique as an alternative

methodology to structured analysis, but still used structured analysis data flow diagrams [7]. Coad and Yourdon combined the best ideas of the existing analysis methods and proposed a more comprehensive and more object-oriented analysis method [26]. However, Coad and Yourdon's relatively new object-oriented analysis method serves as a starting point, and needs tailoring and expanding to suit any specific applications.

The goal of this research is to provide a stand-alone object-oriented requirements specification method which can be applied to real-time systems. A real-time system is defined as any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period [39]. The basic characteristics of a real-time system include concurrent control of separate system components, largeness and complexity of the system, extreme reliability and safety, and continuous change [39]. Using the object-oriented approach, where the concepts of the data abstraction and information hiding is strongly supported, only the objects which undergo the changes should be replaced or upgraded, with the other objects not affected by the upgrade. This advantage assures the modularity and maintainability of the system. Using the

concept of inheritance assures the extensibility of the system by allowing for the definition of new classes of objects from existing ones. A high degree of concurrency can be achieved, since objects are independent entities and may be distributed and executed sequentially or in parallel.

These stand-alone object-oriented specification methods, which will be referred to as Object-Oriented Analysis for Real-Time systems (OOART), will rely only on object-oriented concepts, rather than using such methods as structured analysis. Since the underlying concept of an object-oriented specification is same as that of object-oriented design and object-oriented implementation, the transition is expected to flow smoothly in a software life cycle. The OOART will support complex real-time system characteristics such as concurrent processes and time constraints.

Chapter 2 presents the literature survey on the areas of analysis methods and requirements specification techniques, especially emphasizing the object-oriented approach and real-time systems specification techniques. Chapter 3 states the detailed description of the research. In Chapter 4, a real-time system for multi-process cleaning of silicon wafers is used as a case

study, and the OOART is applied to analyze and represent this real-time system. Chapter 5 evaluates the applied results. A set of criteria will be employed to judge and compare the requirements specification technique in the OOART with other techniques. Finally, Chapter 6 contains some concluding remarks.

CHAPTER II

LITERATURE REVIEW

2.1 Analysis Methods

According to Coad and Yourdon, there are four major approaches to requirements analysis: functional decomposition, the data flow approach, information modeling and the object-oriented method [26]. Each approach has been developed to help in the formulation of a complete, consistent, and feasible requirements specification. The first three of the four approaches have been practiced in the systems development profession for many years and have yielded both successful and unsuccessful results. The object-oriented method is relatively new and will be discussed in more detail.

Functional Decomposition

The underlying strategy of functional decomposition is that each module in the system represents a major step in the overall process [4]. Functional decomposition is recognized with its function, sub-functions and functional interfaces. The focus of functional decomposition is on what processing is required for the system [26]. This method is helpful to break up a large,

complicated process into smaller units; however it has the following limitations [26]:

1. It does not effectively address data abstraction and information hiding.

2. It is generally inadequate for problem domains with natural concurrency.

3. It is often not responsive to changes in the problem space.

4. The problem space understanding is neither explicitly expressed nor verified for its accuracy.

Data Flow Approach

Data flow approach is often described as "Structured Analysis." Structured analysis is probably the most widely used graphically oriented approach and is described in a number of books including DeMarco [32], Gane and Sarson [33], and McMenamins and Palmer [34]. This method emphasizes the graphic depiction of data flows, data transformations and data stores. It incorporates the concept of a context diagram in which the behavior of the entire system is shown, and different levels of data flows and transformations with a data dictionary and process specifications, respectively [26]. The size of the data dictionary can be a problem with

this method. Even though CASE tool support could help somewhat, it is very difficult to grasp the underlying meaning of the data dictionary. Also, the data flow approach has very weak data structure emphasis. The data is always passive and never initiates any action, hence is open to illicit modification [3].

Information Modeling

Information modeling, whose primary tool is the entity-relationship diagram, is very helpful to capture problem space content. The information modeling approach consists of objects, attributes, relationships, supertype/sub-type and associative objects. This strategy is well presented in [28]. The difference between this method and the object-oriented method is that the object-oriented concepts such as services, inheritance, messages and structure are missing in an information modeling approach.

Object-Oriented

The object-oriented method is an approach in which the decomposition of a system is based on the concept of an object. Robson defines an object as a package of information, and the description of its manipulation

[35]. Each object consists of its own data structures and operations that can be carried out on those data structures only by communicating messages. In addition to this data abstraction, the object-oriented analysis includes other important concepts: information hiding and inheritance [26]. Parnas defines information hiding as a principle that each component of a system should encapsulate or hide design decisions about objects, ignoring those aspects of a subject that are not relevant to the current purpose, in order to concentrate more fully on those that are [15]. In daily life, without realizing it, we use the concepts of abstraction and information hiding, and tend to develop models of reality by identifying the objects and operations that exist at each level of interaction. For example, when driving a car, we identify the accelerator, gauges, steering wheel, and brakes (among other objects) as well as the operations we can perform upon them. However, when repairing a car, we identify objects at a lower level of abstraction, such as the fuel pump, carburetor, and distributor. So, the object-oriented approach closely matches our model of reality, and the effect of changing the representation of an object tends to be much more localized; therefore, this method improves

maintainability and understandability of the system. Abstraction and information hiding form the foundation of all object-oriented development [4].

Inheritance is another underlying concept of object-oriented analysis. One object inherits the attributes of another object. Some object-oriented systems provide for inheritance between all objects, but most provide it only between classes, where a class denotes a set of similar but unique objects. A class may be modified to create another class. In this relationship, the original class is called the superclass and the newly created class is called the subclass. A subclass inherits properties or characteristics about its superclass [35]. The concept of inheritance permits a hierarchy of classes. This principle forms the basis for a powerful technique of explicit expression of commonality. Using the concept of inheritance, we can specify common attributes and operations once, as well as specialize and extend those attributes and operations into specific cases [26]. For example, we identify the object "vehicle" and defines some attributes such as a vehicle identification number. A specific case such as the object "truck" will inherit common attributes of the object "vehicle," but might have other specialized attributes which would be appropriate

only for the object "truck." Several object-oriented analysis approaches are described below.

Pressman describes the object-oriented analysis approach in the following manner [36]:

1. The system is described using an informal strategy which is nothing more than an English language description in the form of grammatically correct paragraphs.

2. Objects are determined by underlining each noun or noun clause and entering it in a simple table.

3. Attributes of objects are identified by underlining all adjectives and then associating them with their respective objects (nouns).

4. Operations are determined by underlining all verbs, verb phrases, and predicates and relating each operation to the appropriate object.

5. Attributes of operations are identified by underlining all adverbs and associating them with their respective operations (verbs).

Part of the example from the Pressman text is reproduced in Figure 2.1, in order to illustrate the above object-oriented analysis procedures. First, to determine the objects and their attributes, we underline each noun or noun clause and all adjectives,

respectively. If the object is required to implement a solution, then it is part of the solution space and is entered into the table; otherwise, if an object is necessary only to describe a solution, it is part of the problem space, and is not necessary for the software implementation. The underlined software description using an informal strategy is shown below:

> The software will receive input information from a bar code reader at time intervals that conform to the conveyor line speed. Bar code data will be decoded into box identification format. The software will do a look-up in a 1000-entry data base to determine proper bin location for the box currently at the reader(sorting station). A FIFO list will be used to keep track of shunt positions for each box as it moves past the sorting station.

Object	Attribute	Corresponding operation
Data	Bar code	Decode
Location	Bin	Determine
List	FIFO	Keep track
Positions	Shunt	Keep track
Intervals	Time	Conform
Format	Box id	Decode
Database	1000-entry	Do a lookup

Figure 2.1. Objects and operations.

Next, corresponding operations are added to the table by underlying all verbs, verb phrases, and predicates as follows:

> The software will receive input information from a bar code reader at time intervals that conform to the conveyor line speed. Bar code data will be decoded into box identification format. The software will do a look-up in a 1000-entry data base to determine proper bin location for the box currently at the reader (sorting station). A FIFO list will be used to keep track of shunt positions for each box as it moves past the sorting station.

Like objects, only the operations which belong to the solution space are added to the table. If any significant operation attributes (adverbs) are noted, then they should be entered to the table along with the operations. The definition of objects and operations are an excellent way to begin the analysis of both function and information domains, but specific guidelines and more concrete and effective criteria are desired to define the objects and operations.

Bailin [7] describes an object-oriented requirements specification method which is intended to serve as an alternative to structured analysis for object-oriented

software. Representations used in this specification method are a hierarchy of EDFDs (Entity Data Flow Diagrams) and a set of ERDs (Entity Relationship Diagrams). EDFDs are just like conventional data flow diagrams except that the nodes fall into two categories: entities and functions. Every function must be performed by or act on the entity. This method consists of the following seven steps:

1. Identify key problem-domain entities. Bailin uses a very similar technique as Pressman's to identify domain entities by creating a list of the key nouns and noun phrases from the original statement of requirements. Also, structured analysis data flow diagrams are used to extract nouns from the process names.

2. Distinguish between active and passive entities. An active entity is one that operates on inputs to produce outputs. A passive entity is one that is acted upon.

3. Establish data flow between active entities.

4. Decompose entities (or functions) into sub-entities and/or functions.

5. Check for new entities.

6. Group functions under new entities.

7. Assign new entities to appropriate domains.

Bainlin's method moves smoothly from requirements to design and then to Ada code. However, it seems to be fixed only around Ada tasks, packages, and procedures. Also, it does not address object-oriented characteristics such as inheritance and information hiding [52].

Coad and Yourdon's object-oriented approach is recognized as objects, classification, inheritance and communication with messages [26]. This approach consists of five major steps:

1. Identifying objects: This approach provides a set of characteristics to consider in identifying objects. To find potential objects, look for: structure, other systems, devices, events remembered, roles played, locations and organizational units.

2. Identifying structure: Structures represent complexity in a problem space. Classification structure represents class-member organization, and assembly structure portrays whole-part organization.

3. Identifying subjects: Subjects provide mechanisms for controlling how much of a model a reader is able to consider at one time. So, the amount of complexity confronted by the reader is controlled.

4. Defining attributes: Attributes are data

elements used to describe an instance of an object or classification structure.

5. Defining services: This defines a service as the processing to be performed upon receipt of a message.

The attributes and corresponding services for an object are treated as an intrinsic whole. This analysis focuses on a system's stored data and processing "together." This encapsulation of attributes, and exclusive services on those attributes, provides the stability of the model. This system model results in a tangible, reviewable, and manageable collection of model layers (subject, object, structure, attribute, connection and service) produced during the above five major steps. Figure 2.2 shows an example of Coad and Yourdon's method. An object "Vehicle" contains the common attributes higher in the structure, and the specialized attributes are shown below. The "x" notation indicates attribute override. This method is a relatively new method, and as Coad and Yourdon claim, will continue to evolve in practice. To apply this method to specific organizations or projects, tailoring and expanding the method will be required.

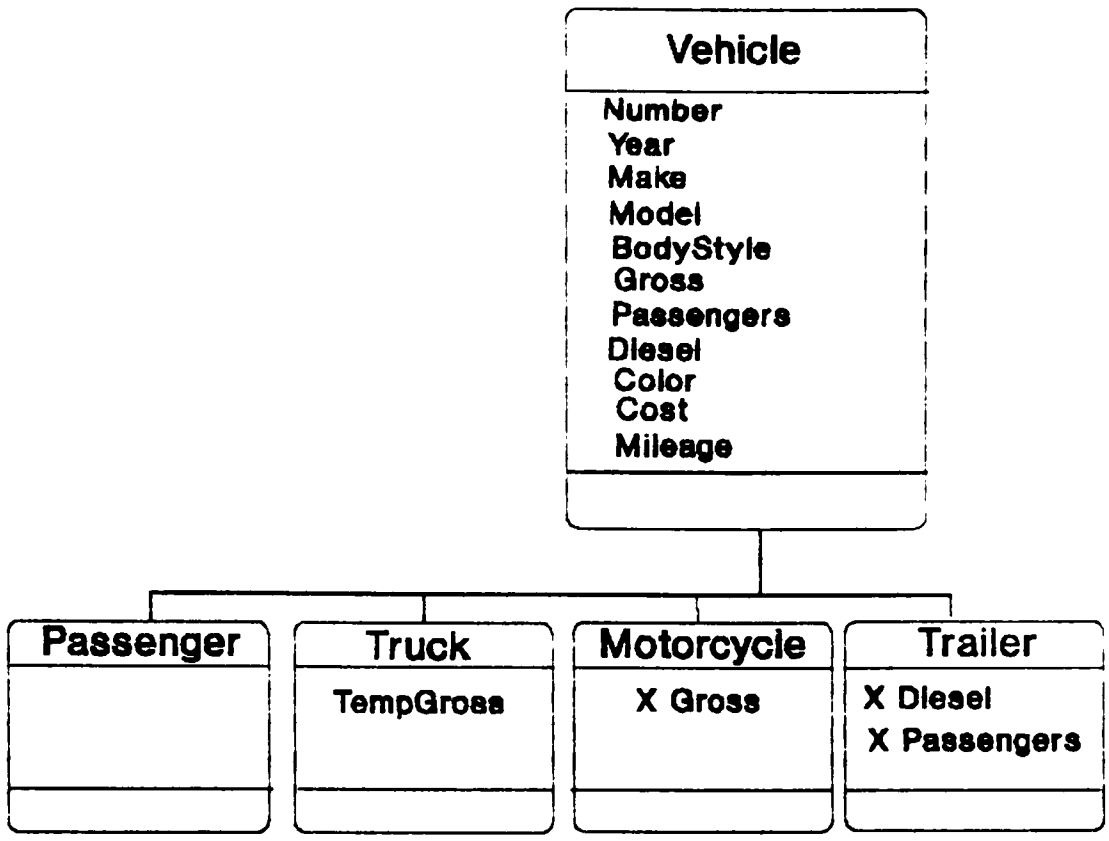


Figure 2.2. Classification structure.

2.2 Requirements Specification Techniques

Various requirements specification techniques have been developed to describe the expected external behavior of the software system to be built. Davis provides a survey of available requirements specification techniques and compares these techniques using the following eight criteria [10]:

1. Understandable to computer-naive personnel
2. Basis for design and test
3. Automated checking
4. External view, not internal view

5. SRS organizational assistance

6. Prototype generation

7. Automatic test generation

8. Appropriate applications.

In this section, each requirements specification technique will be discussed briefly. The table which summarizes each approach and scores them on a scale from 0 (poor) to 10 (excellent) is reproduced from the Davis paper and shown in Figure 2.3. Among the techniques he evaluated, there are five requirements specification techniques suitable for real-time applications: Finite State Machines, Statecharts, Requirements Engineering Validation System, Requirements Language Processor, and PAISLey. In next section, these five techniques for real-time applications will be discussed in more detail. In Chapter 5, the results of this research will be compared to these techniques using the criteria above.

Specification Techniques

Natural language. Most requirements specifications today are written in natural language. Even though natural language can be used as a guide to the development of conceptual models [29], the size of the document and ambiguity of the natural language can lead

Criterion	Natural Language	FSM's	DT's	PDL	SA/RT	Statecharts
1 Understandable to computer-naive personnel	10 Obviously understandable to computer-naive personnel	10 Requires an hour of two of instruction to facilitate understanding by computer-naive personnel	10 Appear in literature regularly without explanation	10 Almost as easy to read as natural language because most of it is natural language	7 See FSM's	5 Many FSM extensions not intuitive to the noncomputer scientist
2 Basis For Design and Test	0 With no formality there is no basis	7 The formal model enables one to unambiguously define intended product behavior and thus serve as a basis for both design and test	7 See FSM's	2 Being little more than natural language, the lack of formality provides no basis	7 The static data-oriented view of a system is helpful for data-intensive applications. See also FSM's	9 With the additional expressive power over FSM's, there is less potential for ambiguity and thus a more formal basis for design and test
3 Automated Checking	0 No processor is available to perform any checking	7 Static structural and behavioral errors can be detected	0 See natural language	1 Static structural errors can be detected	7 See FSM's	9 Static structural, behavioral, and protocol errors can be detected
4 External View, Not Internal View	0 No help provided	3 Needs human discipline in defining the external entities up front	3 See FSM's	0 No help provided	2 For many applications, the data flow view is the design of the software	5 Structural view provided explicitly and independent of behavioral views
5 SRS Organizational Assistance	1 Paragraph and Chapter	2 The machine	2 The table or tree	2 The module	8 Hierarchical definition of transforms and machines	9 See SA/RT; plus more semantic richness of hierarchical alternatives; multiple viewpoints
6 Prototype Generation	0 None	10 Yes, using IDE tools	0 None	0 None	0 None	10 Yes, using I-logix tools
7 Automatic Test Generation	0 None	0 None	0 None	0 None	0 None	0 None
8 Appropriate Applications	Any application where misinterpretation of requirements is acceptable, or at least non-critical	Any real-time application that is small enough to not warrant statecharts, REVS or RLP. For example: small process-control applications	Those particular parts of any application which are decision-intensive	See natural language	Applications with strong data intensity or a need to identify cooperation between FSM's; for example: Most business applications; many process-control applications	Any complex real-time application; especially those with sufficient complexity and criticality to warrant the multiple viewpoints and extra expressive power

Figure 2.3. A comparison of requirements specification techniques.

Criterion	REVS	RLP	SDL	PAISLey	Petri-nets
1 Understandable to computer-naive personnel	7 See FSM's	7 See FSM's	7 See FSM's	2 Many concepts are not intuitive to the noncomputer scientist; hard-to-understand language for same audience	4 Synchrony and Petri-nets appear to be difficult for noncomputer scientists to grasp
2 Basis For Design and Test	9 See statecharts	9 See statecharts	7 See FSM's	9 See statecharts	7 See FSM's
3 Automated Checking	7 See FSM's	7 See FSM's	0 See natural language (potential exists for static structural and behavioral error checking)	9 See statecharts	6 See natural language (potential exists for protocol error checking)
4 External View, Not Internal View	7 Same as RLP, but slightly lower because R-Net is more design oriented than stimulus-response sequence	8 Once external entities are defined, specifier stays naturally in the requirements domain	8 See RLP	3 A common criticism of PAISLey is its design orientation. Zave is not concerned if design is performed during requirements	2 Very difficult to use Petri-nets in the application domain. Many people versed in Petri-nets are designers
5 SRS Organizational Assistance	8 The R-Net plus hierarchies of alphas	7 The stimulus-response sequence; the feature	7 The stimulus-response sequence; the feature	9 See statecharts	2 The net
6 Prototype Generation	7 Yes, using TRW tools plus auxiliary coding	2 Feasibility demonstrated, but not implemented	0 None	10 Yes, using Bell Labs tools	0 None
7 Automatic Test Generation	0 None	10 Yes, using GTE tools	0 None	0 None	0 None
8 Appropriate Applications	Real-time applications that are stimulus-oriented; for example: many defense systems	Real-time applications that are feature-oriented; for example: telephony	SDL was specifically developed for telephone switching systems, but is probably applicable to all RLP applications	PAISLey was developed for complex real-time systems	Those particular parts of any application in which synchrony is critical to the specification of external behavior

Figure 2.3. Continued.

to requirements specifications which are ambiguous and inconsistent, especially for large and complex systems.

Finite State Machines. A Finite State Machine (FSM) is a hypothetical machine that generates an output and changes state in response to an input. The FSM can be in only one of a given number of states at any specific time. Two notations are used to define FSMs: state transition diagrams (STD) and state transition matrices (STM). FSMs have been used effectively for requirements specifications for many applications and serve as the underlying model of another techniques that follow.

Decision Tables and decision trees. A Decision Table (DT) lists all conditions that influence decisions, all the possible decisions, and all combinations of answers in tabular form. The actual required system behavior is then filled into the table. The DT is useful to describe the required external behavior when the system responses to the combinations of more than one condition. A decision tree shows the same information as a decision table but in graphical form.

Program Design Language. The Program Design Language (PDL) consists of free-form English with special meanings for certain key words. PDL is also called pseudocode and is commonly used to specify detailed

designs for software modules. When PDL is used at the analysis phase, the requirements writers tend to overstep their bounds and fall into design.

Structured Analysis/Real-Time. Structured Analysis/Real-Time (SA/RT) reflects two extensions to the Structured Analysis of DeMarco [32]. These extensions add control diagrams and control specs in addition to the standard Data Flow Diagram (DFD). Also, the ability to specify a Mealy model finite state machine description for each control transformer in the DFD is added.

Statecharts. Statecharts are extensions to the FSMs and provides a notation and set of conventions that facilitate the hierarchical decomposition of FSMs and a mechanism for communication between concurrent FSMs. Using the statecharts, it is easier to model complex real-time system behavior without ambiguity.

Requirements Engineering Validation System. The Requirements Engineering Validation System (REVS) is a set of tools that analyzes requirements written in the Requirements Statement Language (RSL) developed using the Software Requirements Engineering Methodology (SREM). SREM was originally developed for performing requirements definition for very large embedded systems having

stringent performance requirements. The REVS uses the R-net as the organizational unit of the SRS.

Requirements Language Processor. The Requirements Language Processor (RLP) is developed for the requirements specification of complex, real-time systems. The RLP uses the stimulus-response sequence as the organizational unit of the SRS where a stimulus-response sequence is a trace of a two-way dialogue between the system under specification and its environment.

Specification and Description Language. The Specification and Description Language (SDL) is a superset of the state transition diagrams. The SDL was developed for the external behavior and internal design of telephone switching systems.

PAISLey. The Process-oriented, Applicative and Interpretable Specification Language (PAISLey) is a language for the requirements specification of embedded systems using an operational approach. Operational means that the resulting specification can be executed, and the resulting behavior would mimic the behavior required of the system to be built. Using PAISLey, the system and its environment are decomposed into sets of asynchronous interacting processes. The major drawback of PAISLey is that the resulting specification is difficult to

understand, and is more like a design than a requirements specification.

Petri-nets. Petri-nets are abstract virtual machines which specify process synchrony during the design phase of time-critical applications. Petri-nets are represented as a graph composed of circles (places) and lines (transitions). Arrows interconnect places and transitions. Black dots (tokens) move from place to place according to a specific rule. Petri-nets are relatively easy to understand and are best used to describe the system where ambiguity should be avoided and precise process synchrony is necessary.

Real-Time Systems Techniques

Among many techniques from the survey done by Davis [10], the next five techniques are distinguished from the other by the applicability of the techniques to the real-time system requirements specification. The FSM can be used for any real-time application that is small enough to be represented with monolithic FSMs. Statecharts, REVS, RLP, and PAISLey are suitable for any complex real-time application. In this research, these requirements specification techniques which are suitable for real-time systems, small or complex, will be discussed in detail.

FSMs. In order to illustrate the use of a FSM, an example of a telephone switching system from the Davis text is reproduced in Figure 2.4 [10]. A circle means a state, a directed arc means the transition between the two states, and the label on the arc means the input that triggers the transition and the output with which the system responds (they are separated by a slash). A FSM is used to model processes which generate an output and change state in response to a finite set of possible inputs whose orders are significant. FSMs can be used to model the system which contains the sequence-dependent operations such as a parity checking of an input stream of ones and zeros, and syntax checking in a high-level language compiler. The FSMs provide a complete and unambiguous specification of all inputs, outputs, and states along with the information needed to specify all error states. Also, FSMs can easily be represented using a decision table [41].

There are two models to describe the behavior of a FSM: The Mealy model and the Moore model. In the Mealy model, all outputs are represented as conditional outputs while all outputs are represented as unconditional outputs in the Moore model. The choice between Mealy and Moore models depends on the particular application. In

the example of Figure 2.4, the Mealy model is used representing all outputs as conditional outputs. FSMs have been used effectively for requirements specifications for any real-time application which is not complex. However, as the system grows complex, the FSM approach become very difficult to understand and modify because of the large numbers of states and transitions involved. The extensions to the FSMs which will be discussed next have been developed for complex real-time applications: Statecharts, REVS, and RLP.

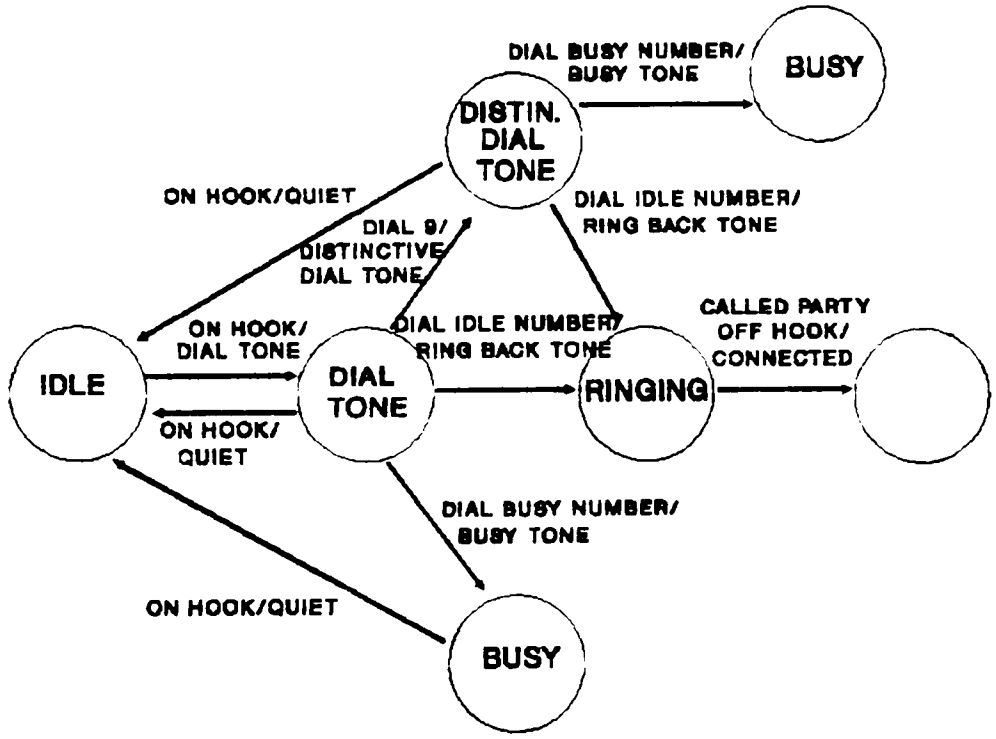


Figure 2.4. FSM example.

Statecharts. Statecharts are an extension to FSMs which provide a notation and set of conventions to model complex real-time system behavior. One of the extensions to FSMs is the concept of the superstate, which can be used to aggregate sets of states with common transitions [10]. An example of the superstate is shown in Figure 2.5 which is reproduced from Davis. In Figure 2.5(b), state S2 is the superstate which have states S21 and S22 combined with the "and" function. The "and" function is represented by a dashed line in the middle of a box and means that on receipt of stimulus i1, the two states S21 and S22 are entered simultaneously. The superstate forms a basis for iterative refinement and successive decomposition of FSMs. Also, the statechart approach has the ability to specify a transition dependent on global conditions and on being in particular state.

REVS. Software Requirements Engineering Methodology (SREM) has evolved in the development of software for ballistic missile defense systems, which are complex, real-time, automated systems with a requirement for high reliability. This approach has resulted the Requirements Statement Language (RSL) and support software REVS. The REVS is a set of support tools which include an interactive graphic package to aid in the specification

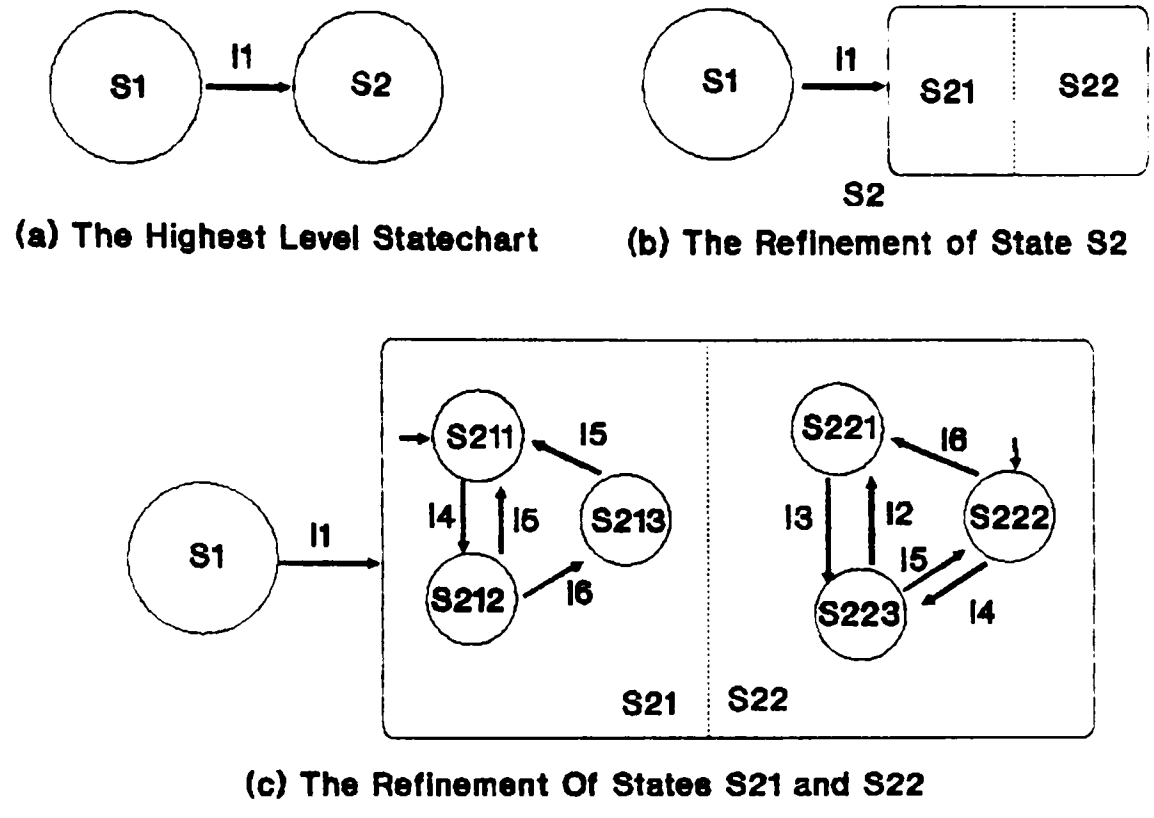


Figure 2.5. Statecharts example.

of flow paths, a static consistency checker, and an automated simulation generation and execution package to aid in the study of dynamic interactions of the requirements [42].

The underlying model of this approach is a highly structured finite state machine. The inputs and outputs are structured into messages that interfaces with external subsystems. Each state is structured into sets of information about objects that are known to the data processor. Requirements networks (R-nets) are used to structure the description of the processing. Each R-net

specifies the transformation of an input message and current state into a set of output messages and an updated state [43]. An example of an R-net is reproduced from Davis in Figure 2.6. Using the R-nets, accuracy and response time performance requirements which are critical in real-time systems can be specified.

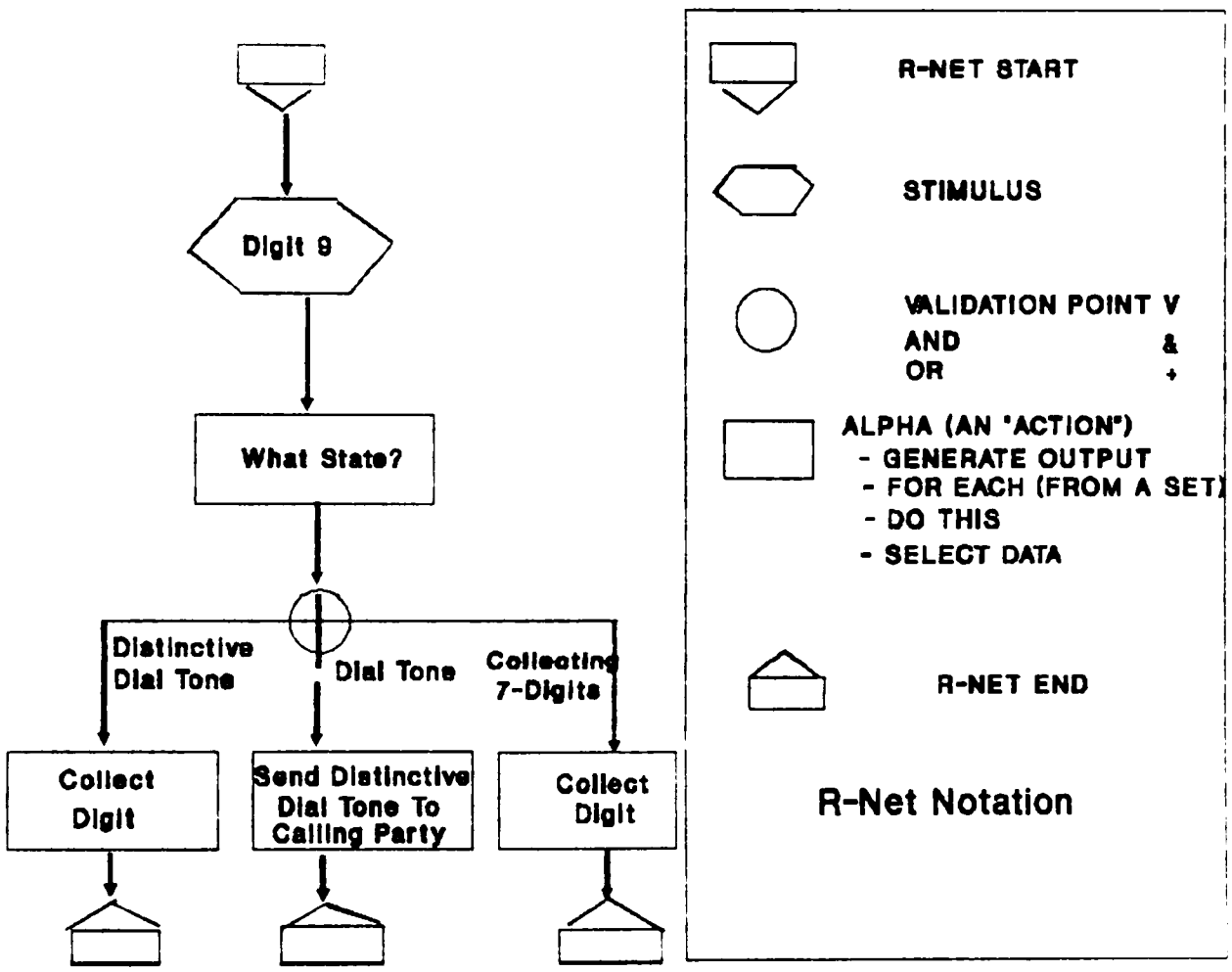


Figure 2.6. REVS example.

RLP. The RLP uses the stimulus-response sequence as its organizational unit. The system must "respond" on receipt of a particular "stimulus." For any real-time system, the concept of timing constraints are applied to the stimulus-response such that the system must respond within a specific period of time receiving a particular stimulus. The RLP accepts and processes requirements documents written in a wide variety of requirements languages. The main inputs and outputs of the RLP are shown in Figure 2.7 [45]. The language definition tables define the lexical, syntactic, and semantic characteristics of a requirements language. After these tables are defined, a user of the RLP can select an appropriate language, write requirements using that language, and run RLP. The RLP outputs an integrated system description in the form of an extended FSM and reports any inconsistency, ambiguity, and incompleteness in the requirements. The FSM description can be used later to drive automatic test generation, system simulation, and automatic software synthesis.

PAISLey. The PAISLey is a language for the requirements specification of an embedded system which is characterized by asynchronous parallelism and urgent performance requirements [46]. Using PAISLey, a

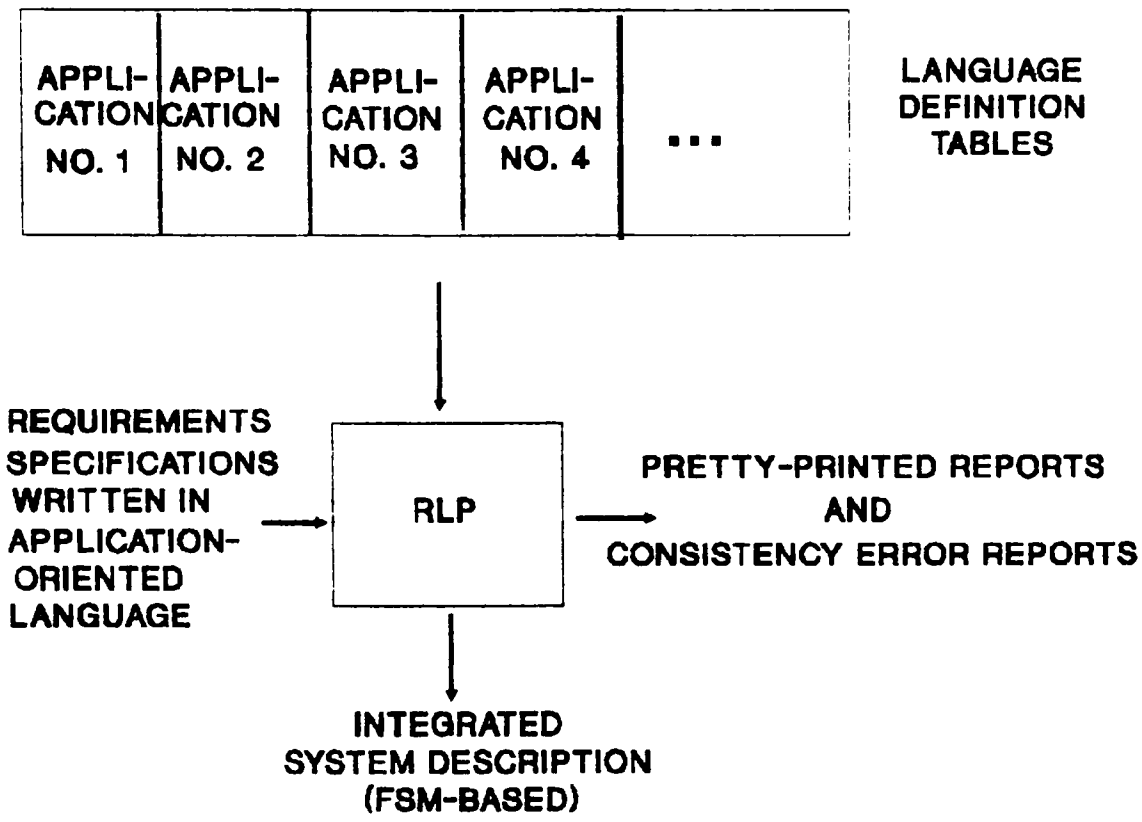
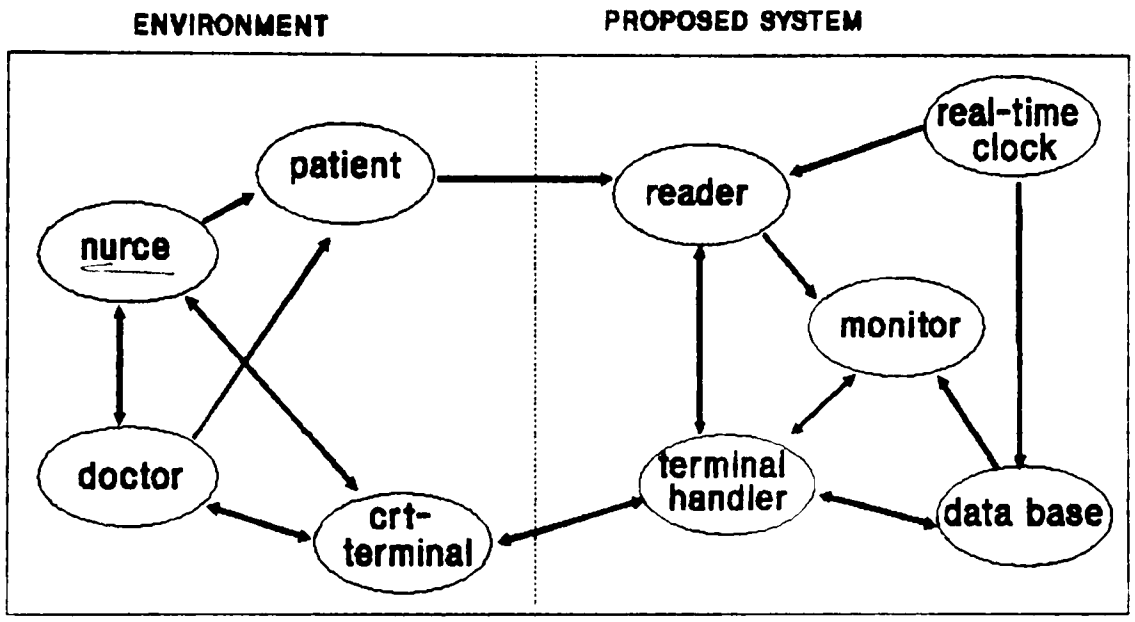


Figure 2.7. RLP example.

requirements specification is an executable model of the proposed system. As shown in the example of a patient-monitoring system (Figure 2.8) which is reproduced from Davis, the system and its environments are decomposed into sets of asynchronous interacting processes: five processes for the system and four processes for its environment. Then, each process is defined further by declaring the state space, successor function, and exchange function. In Figure 2.8, partial declaration of the state space and successor functions for the process

"patient" is shown [10]. A specification in PAISLey can retain all parallelism inherent to the problem space of the system. In the execution of the PAISLey specification, all computations are encapsulated and can be executed no matter how incomplete they are. Also, all forms of inconsistency can be detected by automated checking [47].



(patient-cycle: PATIENT_STATE --> PATIENT_STATE,
patient-check-cycle: ALERT_STATUS --> ALERT_STATUS
retrieve: DATABASE x REQUEST -->DATABASE x VITAL_SIGN_TOLERANCE
sense: PATIENT_STATE --> ACTUAL_VITAL_SIGNS
check: VITAL_SIGN_TOLERANCE x ACTUAL_VITAL_SIGNS --> ALERT_STATUS
update: DATABASE x ACTUAL_VITAL_SIGNS --> DATABASE)

Partial Declaration of Processes and their Successor Functions

Figure 2.8. PAISLey example.

2.3 Approach to Integrate Object-Oriented with Structured Analysis Method

Ward claims that there is no fundamental opposition between real-time structured analysis/structured design and object-oriented design, and proposes a method to integrate object orientation with structured analysis and design [6]. He shows that objects as instances of abstract data types can be represented in structured analysis/structured design by grouping of lower level transformations and stored data into higher level transformations. The data-specification scheme of real-time structured analysis/structured design must be extended to reflect the instance/type distinction for data items. The concepts of inheritance can be incorporated into structured analysis/structured design by embedding operations and data for higher level objects in the transformations representing the lower level objects. Also, the convention for flow convergence/ divergence and the hierarchical numbering scheme for levels of schemas should be changed to reflect the concepts of inheritance.

Even though he claims that real-time structured analysis/structured design can, with modest extensions to the notation and to the model-building heuristics, adequately express an object-oriented design, the

integration requires another extra step to develop an object-oriented model in the development activities for a software system. As the system grows large and complex, it will become very difficult to trace back to the functions and data from different levels and group them together to incorporate the concepts of objects or inheritance. So, if the object-oriented design and implementation are foreseen in the development of the system, it is ideal to apply the same concepts to the requirements analysis.

2.4 Summary

The real-time requirements specification techniques described in the literature lack the following features which are needed for an object-oriented real-time methodology:

1. They do not support all characteristics of the object-oriented approach: data abstraction, information hiding, and inheritance.

2. They do not model complex real-time system behavior.

3. They do not support the distributed multiple processes without using any centralized, global controller.

CHAPTER III

OBJECT-ORIENTED ANALYSIS FOR REAL-TIME SYSTEMS (OOART)

The goal of this research is to produce a stand-alone object-oriented analysis methodology which can be applied to real-time systems to generate a complete, consistent, and reliable requirements specification. This chapter states the detailed description of the research. Section 1 deals with the definition of objects, and Section 2 shows how each object is structured internally. Sections 3 and 4 explain how the message passing is handled, and how the analysis method is represented in the requirements specification, respectively.

3.1 Definition of Objects

Basic Concepts of Objects

In an object-oriented approach, the system is viewed as a collection of objects. Analysis of the system is accomplished in terms of these objects. Objects and relationships between objects which are identified and formally represented in the analysis phase serve as the basis for the next software development phases--design

and implementation. The object-oriented software life cycle tends to eliminate the distinct boundaries between the analysis, design, and implementation phases because the items of interest in each phase are the same: objects [53]. To achieve the successful development of the object-oriented software, it is crucial to identify and define the "right" objects for the system.

As described earlier in Chapter 2, objects are packages of both data and operations that can be carried on that data. Data and process are encapsulated into an object as an "intrinsic whole" [26]. This encapsulation of data attributes and operations on that data makes the effect of changes on the system over time to be much more localized, hence makes the system more stable. From the viewpoint of a systems analyst, an object is an abstraction of the real world, representing one or more occurrences of meaningful entities in the problem space. The system will be structured around the objects that exist in our model of reality. The most significant advantage in defining objects as an abstraction of the real world is that the activity makes the systems analyst to understand the problem space at hand. As Booch states, the steps in abstraction "requires a great deal

of real world knowledge and an intuitive understanding of the problem" [4].

Another important characteristic of objects is that one object can inherit the attributes/operations of another object. This can be accomplished by introducing a concept of a class which denotes a set of similar but unique objects. Each object is a unique instance of some class and a class may be modified to create another class, establishing a relationship of subclass and superclass between the classes involved. A subclass not only inherits the attributes and operations of the superclass, but also can modify the existing attributes/operations, add new ones, or hide some of them from the superclass. This concept of inheritance allows a hierarchy of classes and reduces the work of defining the similar objects again and again. Also, this activity of defining objects in terms of classes matches with the basic human method of organization. According to Coad and Yourdon [26], there are three basic methods people use to organize problem space:

1. The differentiation of experience into particular objects and their attributes.

2. The distinction between whole objects and their component parts.

3. The formation of and the distinction between different classes of objects.

Communication between objects is done by message passing. An object sends a message to another object to perform an operation. When the object receives a message from another object, it performs the requested operation if all the necessary conditions are met. Otherwise, it sends an error message back to the object which sent the unsuccessful message. Therefore, each object has the capability of receiving and sending operation invocation and error handling messages. The notion of a "method" is introduced from Smalltalk, an object-oriented programming language and from which the term "object-oriented" originated. A method denotes the response by an object to a message from another objects. The activity of a method may activate the predefined operations or pass the messages to other objects to invoke one of its methods. The granularity of objects is related to the amount of inter-object communication. During the analysis phase, the granularity of objects is determined, and the interfaces (communication) between objects are formalized. The object-oriented approach for problem definition and partitioning is then applied to

the analysis. The strategies to identify the objects are discussed below.

Strategy to Identify the Objects

Identification of the objects is the first step in the object-oriented analysis. This is the process of finding the entities that represent the best abstraction for what the system models. However, there is no simple mechanism that can be applied once to provide the best definition and partitioning of objects. The identification of objects (including the identification of attributes and operations for the objects) involves a number of local iterations until the satisfiable abstraction of the system is obtained. However, this iterative process should not be considered as the waste of the analyst's time and energy, but is regarded as the process of gaining sufficient and solid understanding of the problem space. Even though the intuition and experience play the important roles in this process, there are methods which can help the analyst to identify these objects; these methods are described below. The following strategy to identify the objects and relationships between them is based on methods developed by Booch [4], Seidewitz and Stark [3], and Coad and

Yourdon [26]. This particular strategy is derived to combine the best elements of each method and develop the best abstraction of the problem space.

Recognize the preliminary objects. The first step is to look at the problem space and recognize the major entities by differentiating the problem domain into particular objects that should play their roles and have attributes in our model of reality. The preliminary objects identified in this step usually can be derived from the nouns which are used in describing the problem space. Although not every noun will be turned into the objects, giving special attention to the noun phrases is a reasonable starting point to distinguish the potential objects of the system.

Identify the attributes and operations. The second step is to recognize the attributes and operations for the preliminary objects derived from the previous step. Attributes should be able to describe the every possible state of the object, and operations should provide the necessary processing of the object. A common set of attributes and operations is then identified for each object. The appropriate granularity of objects is considered here, with larger objects are decomposed into smaller objects and vice versa. This process of

filtering out the preliminary objects to get the better set of objects goes through a number of iterations between the first and second steps.

Build the layers of abstraction. The third step is to build the different layers of abstraction by using the classification and assembly structures. Each layer will contain a collection of objects with restricted visibility to other layers. The purpose of this process is to represent problem space hierarchy and to establish the visibility of each object in relation to other objects. Classification structure provides the partitioning of a problem space into the sets of generalized and specialized objects. The objects which have the more generalized (common) operations and attributes are put at a higher level, and the specialized ones at a lower level. For example, the object "Vehicle" is put at a higher level with the common attributes and operations, and the specialized objects such as "Passenger Car," "Truck," or "Trailer" are placed below. This structure supports the concept of inheritance. The assembly structure shows a whole and its components. Objects are partitioned into its component objects according to the appropriate granularity of objects. For example, an object "Vehicle" can have the component

objects such as "Engine," "Brake," or "Body." This structure reflects a real-world model of the problem space. Using the classification and assembly structures, a system is partitioned into different layers of abstraction. An object diagram such as in Figure 3.1 can be used to represent the layers of abstraction [4]. Any layer in a higher level can use any operations in the layers below, but not any operation above that layer. This assures that all objects in one layer do not need to know about the objects on the higher layers. The main advantage of this topology of connections is that it generates the loosely coupled objects by reducing the coupling between objects.

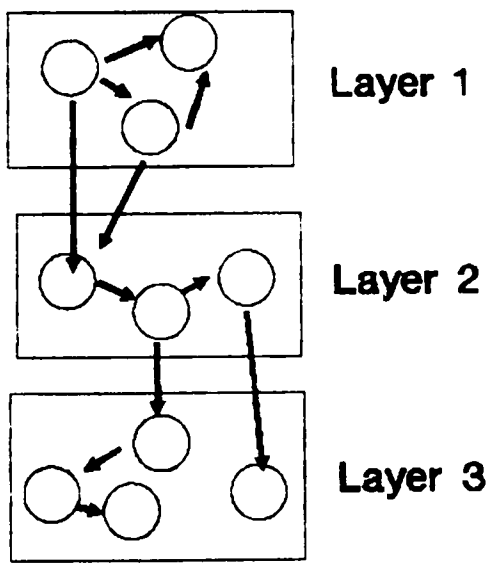


Figure 3.1. Object diagram showing the layers of abstraction.

Establish the interfaces. The final step is to establish the interfaces between objects. Communication between objects is done by message passing. This process validates the topology of objects defined previously, or modifies the layers and objects to support the more feasible and efficient message passing between objects. Since each object may be independent, there exist many threads (flow of control) simultaneously running through the system. This activity supports the concurrent control which is one of the characteristics of the real-time systems.

As a summary, the following steps are highly iterative steps until the final objects and relationships are established:

1. Study the problem space and recognize the preliminary objects.

2. Identify the attributes and operations for objects.

3. Build the layers of abstraction using the classification and assembly structures.

4. Establish the interfaces between objects.

3.2 Detailed Description of Objects

Structure of an Object

The internal structure of an object is discussed in this section. Each object includes the data (both the data that make up the attributes of the object and the data for a message queue), the methods that define the operations on that data, and other elements such as the utility routines, storage for the data structures defined, and protection, etc. However, all this information is encapsulated, with only the entry points defined at the interface are visible to the other objects. The entry points of an object include a unique identifier and the methods. That is, when other objects are sending messages to an object, they use the unique identifier to recognize the object with which they want to communicate, and specify the method to request the operation invocation to that object. Figure 3.2 shows the internal structure of an object, and each element of an object is explained in the following paragraphs. However, it should be pointed out that it is the external view of an object that is emphasized in the analysis phase. It is also assumed that each object has an access to the source of the accurate knowledge of time (the

clock). The clock will be explained in detail when the time constraints are discussed.

Identifier. Each object has a globally unique identifier that can be used by other objects when they are sending messages to invoke the operations on it. An object dictionary is employed to keep track of the identifiers defined and to avoid the duplicate names.

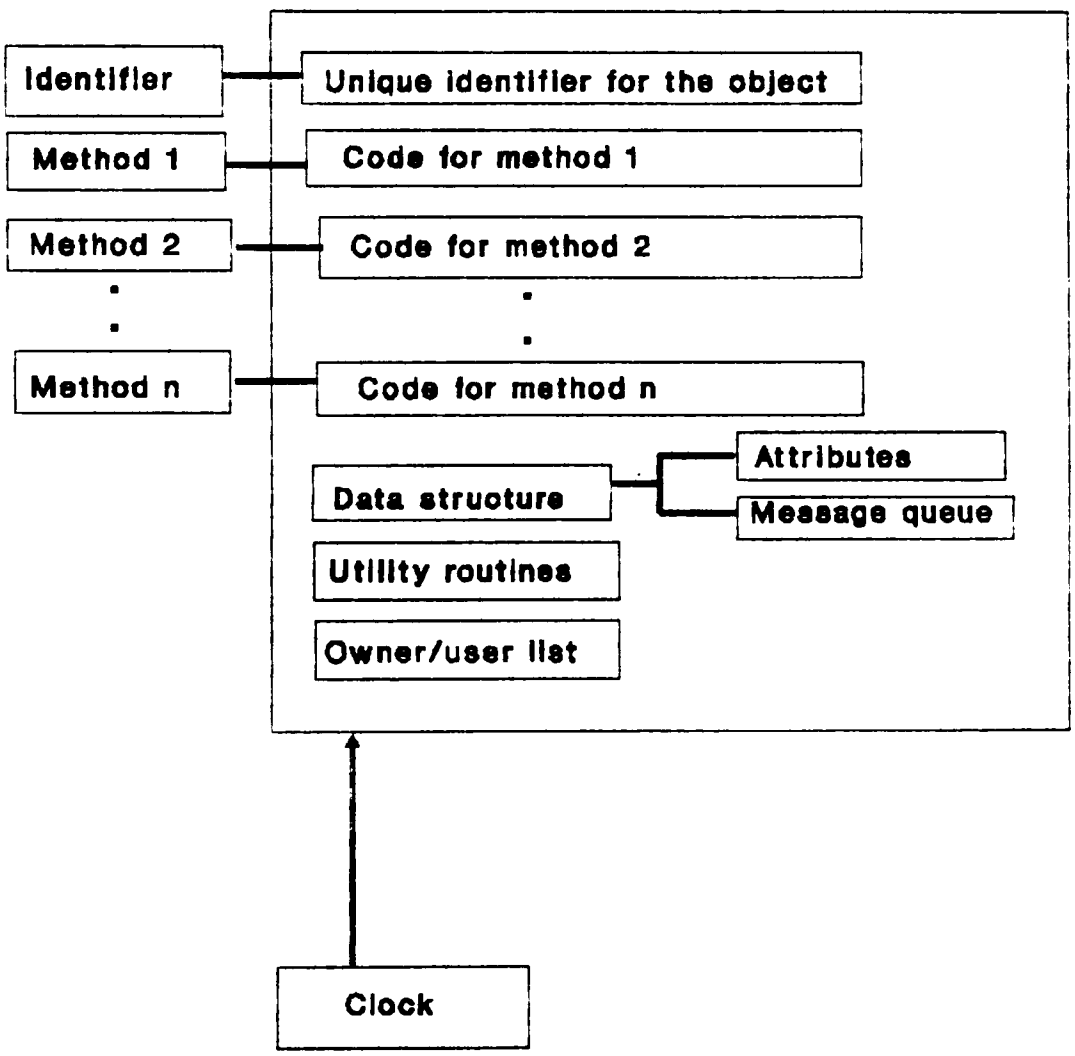


Figure 3.2. Internal structure of an object.

Methods. All the operations that can be performed on the data should be defined and implemented in an object's methods. Methods (which were originally defined in Smalltalk) determine an object's behavior and performance. They are like function definitions in structured analysis. The methods are initially inactive, and upon receipt of the invocation request message (via message queue), the requested method becomes active and the code which implements the operation is executed. After the execution is completed, a message to notify the completion (notification message) is sent to the object which has sent the request message, and the method becomes inactive again until another invocation request is received. All the incoming messages to an object are delivered to the specified method through the message queue. The reason for employing the message queue is to take care of the request which is the most important and urgent (has the highest priority value) first at any specific moments. When an object receives a message from another object, the message contains a specific method name for operation invocation and any parameters (if needed) along with other information such as the priority value and time constraints. This time constraint is a very important characteristic of real-time systems. (The

mechanism to handle the time constraints will be discussed later in detail.) Also, each operation in the methods should be associated with a worst case execution time, so that this information can be compared with the requested time constraints. Before being stored into the message queue, the requested method is evaluated to make sure that the method exists in the object and that the received parameters are compatible with the predefined parameters in the object. If the requested method is not found, or the parameters are not matched, an error message is sent back to the object which has sent the message. Otherwise, the message is stored into the message queue. The code for this message manipulation (receive, evaluate, and store) is implemented in one of the utility routines. Whenever the message queue is not empty and all the methods are inactive, the message with the next highest priority value at that time is retrieved from the message queue and is delivered to the specified method. The utility routines also include an operation which performs this message retrieval. Operations which are defined in the methods may change the value of the attributes of the object, just evaluate them, or invoke operations on other objects by sending the messages to them.

Data structures. Data structures are defined to include all the data attributes of an object, as well as the message queue. In an object-oriented approach, the operation and data are not separable. Any data which the operations in the methods use should be included in the data structure. The message queue is itself an object. The purpose of the message queue is to receive the messages from other objects and distribute them in the order of higher priority values. A detailed explanation of the message queue is found in the following section.

Utility routines. Utility routines include any subroutines which are necessary to perform the message and object manipulation other than the operations defined in the methods. Utility routines are same for all objects. Therefore, when an object is created, they can be inherited from a superclass object instead of being redefined for each object. The following subroutines are included in the utility routines:

1. The subroutine which receives the incoming messages, and stores them into the message queue, or sends back to the originated object if any errors are found.

2. The subroutine which retrieves the message with the highest priority value.

3. The subroutine which keeps track of other objects which are using the object. (The user list contains the identifiers of the objects which might use that object.)

4. The subroutine which performs exception handling. When an error (including a failure to meet time constraints) is detected, the exception handling routine is invoked to take care of the condition and to resume the normal execution of the system.

5. The subroutine which takes the time constraints from the message, checks that the starting conditions are met, and enforces the constraints on the requested method so that the execution should take place and be completed within a specified finite time.

Owner/user lists. Owner/user lists are kept to avoid the undesired deletion of objects which are still needed by other objects in the distributed multiple-user systems. When an object is created, the ownership of the object is declared, and only the owner can delete the object, provided that there are no users attached to the user list. The owner/user list is manipulated by one of the utility routines.

Structure of the Message Queue

When an object receives messages from other objects, those messages are sent to the message queue (by one of the utility routines, as explained in the previous section), and are distributed in the order of the highest priority value at the current time. The message queue is itself an object. The message queue object, which is defined in the data structure of the main object, is automatically created when the main object is created, and deleted when the object is deleted. Figure 3.3 shows the internal structure of the message queue object. Each object has one and only one message queue object for its own message traffic control. Therefore, a one to one mapping exists between an object and its message queue. To support operations that allow the item containing the largest value to be easily retrieved and deleted, a priority queue will be used to implement this object [51]. The priority value reflects the urgency with which the message should be taken care of. Many factors should be considered to decide the priority values for the messages. How to set up and deal with priority values will be discussed later in detail. The task of the message queue is to receive all incoming messages and send back the message with the highest priority value at

a time to the object with which the message queue is connected. All the actions regarding time constraints and exception handling will be done by the object which contains the message queue, and not by the message queue itself.

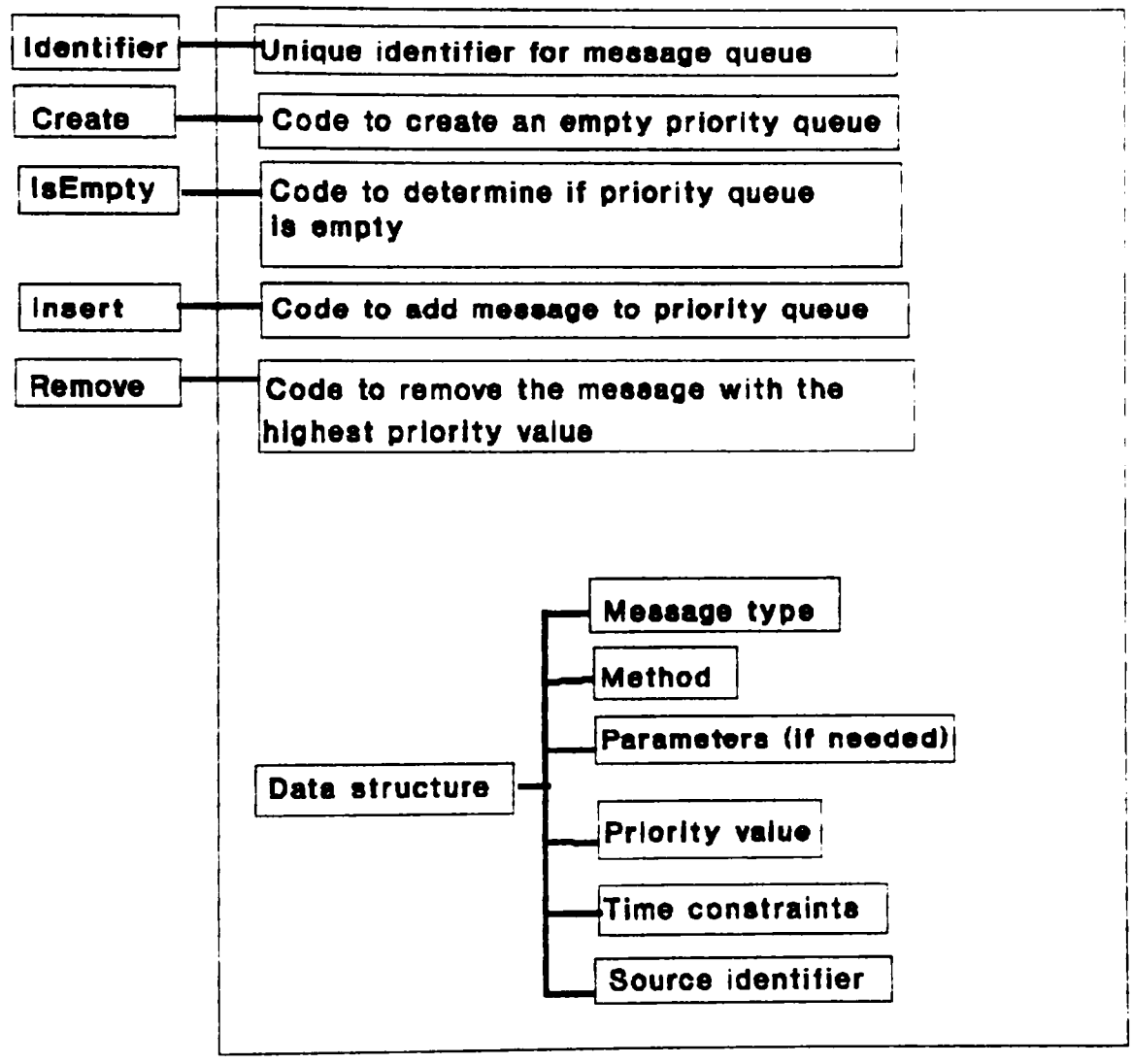


Figure 3.3. Internal structure of the message queue.

Identifier. Each message queue object has a unique identifier that matches with the object to which the message queue is connected. When an object is created or deleted, its message queue object should also be automatically created or deleted. The identifier for the message queue object can be implemented as a pointer to the main object.

Methods. The message queue object supports the following operations in the methods:

1. "Create" to create an empty priority queue. It is invoked automatically when the object is created.

2. "IsEmpty" to determine if priority queue is empty.

3. "Insert" to add a message to priority queue.

4. "Remove" to remove the message with the highest priority value and send it to the main object.

Data structure. The definition of the data structure of the message queue should include the attributes of a message such as the message type, requested method, parameters if needed, priority value, time constraints, and the identifier of the object which has sent the message. All these elements are discussed in detail later when the format of a message is explained. In the analysis phase, the decision of how to

implement the priority queue does not have to be made. (However, the priority queue can be implemented as a binary search tree or heap.)

Examples of Objects in the Ada Specification

Ada is one of many languages that are better suited than others to the application of object-oriented development. The following Ada packages are included to give some idea of how the internal structure of the object and the message queue can be implemented. Objects are denoted by packages, and the attributes of an object reside in the body of a package. The package specification gives the interface to the outside world, and the body of the specification gives the hidden details of the operations in the methods. Common operations and objects are imported from a program library using a with clause. A library unit may be a subprogram specification such as a set of utility routines or a package specification such as the message queue object. To support parallel activities, a task may be used with a rendezvous which allows tasks to interact with each other. The methods of an object will be executed one at a time, but the subprograms in the utility routines may be executed in parallel. For

example, while receiving the incoming messages (Receive), the message with the highest priority value is retrieved from the message queue (Retrieve), and the requested method is executed. The following example shows how an object will be denoted by the Ada package.

Example 3.1. Ada package for an object.

```
-- import message queue and utility routines
with Message_Queue_Identifier;
with Utility _Routines; -- implemented as tasks

package Object_Identifier is  -- specification
   procedure Method1;
   procedure Method2 (Parameter: P_Type);
end Object_Identifier;
package body Object_Identifier is  -- body
   Attributes: Attribute_Type;
   Owner_List: array (INTEGER range 1 .. N) of
               Identifier;
   User_List : array (INTEGER range 1 .. N) of
               Identifier;
   procedure Method1 is
   begin
      -- code for method 1
   end Method1;
   procedure Method2 (Parameter: P_Type) is
   begin
      -- code for method 2
   end Method2;
begin
      -- use imported utility routines
      use Utility_Routines;
      Receive(In_Message: Message_Type);
      Exception_Handling;
      Highest_Priority_Message := Retrieve;
      Maintain_Owner_List;  Maintain_User_List;
      Get_Time_Constraints(In_Message:Message_Type);
      case Highest_Priority_Message is
        when Request_Method1 => Method1;
        when Request_Method2 => Method2;
        when others => null;
      end case;
end Object_Identifier;
```

The message queue is also denoted by the package and stored as a library unit as follows.

Example 3.2. Ada package for the message queue.

```
package Message_Queue_Identifier is
   type Message_Type is limited private;
   procedure Create;
   function IsEmpty return Status;
   procedure Insert (Message: Message_Type);
   function Remove return Message_Type;
private
   type Message_Type is
      record
        M_Type: INTEGER;  -- operation invocation,
                          -- notification, or
                          -- error handling type
        Method: INTEGER;  -- requested method
        Parameter: ...    -- parameters if needed
        P_Value: INTEGER; -- priority value
        Time_Constraints: T_Type;
      end record;
end Message_Queue_Identifier;

package body Message_Queue_Identifier is
   Message_Queue: Heap_Type;
   procedure Create;
   begin
    -- code for Create
   end;

   function IsEmpty;
   begin
    -- code for IsEmpty
   end;

   procedure Insert(Message: Message_Type)
   begin
    -- code for Insert
   end;

   function Remove;
   begin
    -- code for Remove
   end;
end Message_Queue_Identifier;
```

Adding Time Constraints to Objects

Time constraints are one of the most important characteristics of real-time systems. Real-time systems impose strict requirements on the timing behavior of the system. That is, if the system's response does not occur within a specified time, it is considered a system failure. In the analysis phase of real-time system software development, the problem of how to implement time constraints does not have to be addressed, but the problems of how to specify the time constraints accurately and unambiguously should be considered and formalized [37]. Time constraints are added to the object by message passing, and handled by the utility routines of the object. When an object sends a message to other objects for operation invocation or error handling, the message should contain the time constraints for the specified operation. These time constraints are compared against a worst case execution time which is attached to each operation to determine whether the requested operation can be finished within the specified time. In this section, the strategies to represent and assign time properties to the objects are discussed.

Timing properties enforce time bounds on the computations and relations between objects. After a

method receives an operation invocation message, the object should satisfy any necessary starting conditions and start executing the requested operation by the specified start time. Also, the execution should be completed before its stop time (deadline). In the OOART, the following elements are included in the message to represent the time constraints: Starting conditions, Begin time, and Stop time.

Starting conditions. This starting conditions state any necessary conditions that should be satisfied before the object executes the requested operation. Temporal operators and relations shown in Figures 3.4 and 3.5 are used to state these temporal conditions and properties [49, 55].

Begin time. Begin time can either state the specific time point by which the object should start executing the specified operation, or state the specific condition to be satisfied to begin the execution. A real number is used for a time point that represents the occurrence of an instantaneous event, and a set of real numbers such as <t1, t2> is used for a time interval that represents a contiguous time period. Temporal operators and relations are used to specify the conditions. If the

temporal relation is the same as the starting condition, it can be omitted.

Notation	Meaning
⊙ p	At the next instant, p will be true.
p U q	p is true until q is true, and q will eventually become true.
◇ p	Eventually, p will become true.
□ p	Now and always in the future, p is true.
⊖ p	At the previous instant, p was true.
p S q	p has been true since q was true.
⧫ p	sometime now or in the past, p is true.
⊟ p	Now and always in the past, p was true.
p -> q	if p is true, then q is true.

Figure 3.4. Temporal operators.

Notation	Meaning
A = B	Two intervals are equal.
A < B	A precedes B.
A ⇑ B	After A finishes, B meets A immediately.
A ∅ B	A and B are overlapped.
A ↑ B	A starts with B, and finishes first.
A << B	A occurs during B.
A ↓ B	B starts first, and ends with A.

Figure 3.5. Temporal relations.

Stop time. Stop time can either state the deadline before which the object should complete the specified operation, or state the specific condition to be satisfied to stop the execution. The means of representation are same as those of Begin time.

Temporal notations. The temporal operators in Figure 3.4 are used in temporal logic which is a well-developed branch of modal logic. Temporal logic has been used for the specification and verification of program behavior [55]. For example, the temporal operators can be used to denote a starting condition in which condition 1 is to be true, and immediately after condition 1 is true, condition 2 should be true as follows.

◇ condition 1

condition 1 -> (condition 2 S condition 1)

In addition to the temporal operators, the temporal relations in Figure 3.5 will be used to support the timing descriptions of various ordering relations. These notations were developed by Levi and Agrawala [49] for representation of the temporal relations. Let A and B be the contiguous time intervals. For example, event 1 occurs during the time interval 1 and event 2 occurs during interval 2. Both interval 1 and interval 2 are represented as sets of real numbers. Then, using the

notations of temporal relations, it can be denoted that event 1 and event 2 should start at the same time, and event 1 should be over before event 2 is finished with the expression event 1 ↑ event 2. At the design and implementation stages, an appropriate data structure will be employed to implement the time constraints, and the specific algorithms will be applied to guarantee to meet the constraints. But, at the analysis phase, the above information is specified for every operation invocation, integrated with the other contents of the message, and sent to the object to specify the time constraints.

Clock. Each object is assumed to have an access to a clock which is the source of the knowledge of time. There are three categories of clock systems: (a) central clock systems, (b) centrally controlled clock systems, and (c) distributed systems [49]. Any clock system can be employed to assure the accurate knowledge of time. In central clock systems, the whole system receives the time knowledge from one accurate clock. Centrally controlled clock systems have the master clock and slave clocks. Master clock polls slave clocks and corrects the differences. In distributed systems, each object has its own clock and updates the differences after receiving the time from other objects.

3.3 Message Passing Techniques

Communication between objects is done by message passing. A message allows for only one channel of communication between objects. Message passing can be done either synchronously or asynchronously. In the synchronous message passing environment, processes must poll and wait for their turn to send messages. Since each object has its own message queue and is ready to receive the incoming messages at any time, the receiver object does not have to wait for the message in order to receive it. Also, the sender object does not have to wait for ensuring the correct arrival of the message at the receiver object because an error handling message will be sent to the sender object if any errors are occurring. Therefore, the message passing technique for the OOART can be based on an asynchronous environment, where several loosely coupled processes communicate with one another asynchronously. This section will explain how the message is structured and processed to make the objects in the system communicate with each other.

The Role of the Messages

A message is a communication between two objects. An object may send a message to another object to request

the operation invocation or to obtain the current status of an operation. Since communication between objects is done only by message passing, shared data areas can be greatly reduced. In a case of the distributed multiple computer systems connected by a Local Area Network (LAN), this reduction helps in controlling the processes [30].

As mentioned earlier, when an object receives a message from another object, it is placed into the message queue before being distributed to take care of the appropriate request. Even though heavy communication traffic is involved in this message-driven environment, the number of messages in the message queue is expected to be small because most messages can be processed quite rapidly if a high degree of concurrent processing is achieved.

Messages also carry the information which is necessary to process the requested operations in the method. Parameters are passed into and out of the object through the message. The number and types of parameters should match with those specified in the method of the object. In the next section, the format of a message will be discussed in detail.

The Format of a Message

Each message consists of the following elements: destination identifier, message type, message contents, parameters, priority value, time constraints, and source identifier.

Destination identifier. This element is the identifier of the particular object to which the message is addressed. An object may send a message to any other object whose identifier is available. As mentioned in the section on the definition of objects, large object-oriented systems tend to be built in layers of abstraction. Each layer has restricted visibility to other layers.

Message type. There are three categories of message types: operation invocation messages, error handling messages, and notification messages. The message type indicates which category the message belongs to. The operation invocation messages, which carry the request to execute the particular methods of the destination object, are the most common messages. When an object sends an operation invocation message to another object, the message should contain a name of the method in which the desired operations reside. An error handling message is sent to the object when appropriate. For example, when

an object sends an operation invocation message with its parameters, and the destination object detects errors in the received parameters, the destination object sends an error handling message back to the source object. The notification message notifies the source object that the requested operation has completed its execution. The notification message can also carry any return values of the executed operations.

Message contents. This element contains different kinds of information, depending on the message type. For an operation invocation message, this field has the name of the particular method in the destination object. For an error handling message, the status of the error condition is specified. Any return values of the executed operations are delivered through the message contents for a notification message.

Parameters. The parameter block contains any parameters which the method of the destination object uses. A message should supply a correct parameter block which contains the same number and type of parameters as specified in the methods of the destination object.

Priority value. The priority value indicates the urgency of the message. The messages in the message queue will be retrieved and taken care of in the order of

the highest priority value. The priority value can be set to any integer, with the higher the integer, the greater the priority. A message with the higher priority value is specified as a more important and urgent request; a message with the lower priority value is less urgent.

Time constraints. This field contains the time constraints which will be enforced on the requested operations in the methods. As explained earlier, the time constraints block has three elements in it: starting condition, begin time, and stop time.

Source identifier. This element is the identifier of the object which is sending the message. The source identifier is included, so that the destination object can communicate with it.

How to Initialize the System

Using the OOART, the system is specified as a collection of objects. Among these objects, an object which we named as "supervisor object" performs the specific tasks. This object acts like a supervisor at the work place as shown in Figure 3.6. A supervisor for the manufacturing department gives some assignments to his workers and lets them start their jobs

(initialization). While the workers are doing their jobs (communicating with each other, and exchanging the materials if needed), the supervisor does not interfere with them unless something unexpected happens (and error handling required) or the workers have any questions or requests (global control request) for the supervisor. Also, the supervisor can stop the manufacturing processes (shutdown).

In a distributed computer system, the process modules may reside in the independent computers which are connected to a host computer by a LAN. The supervisor object resides in a host computer. Even though the supervisor object manages and gives global coordination to the objects in the system, not all messages must pass through the supervisor object.

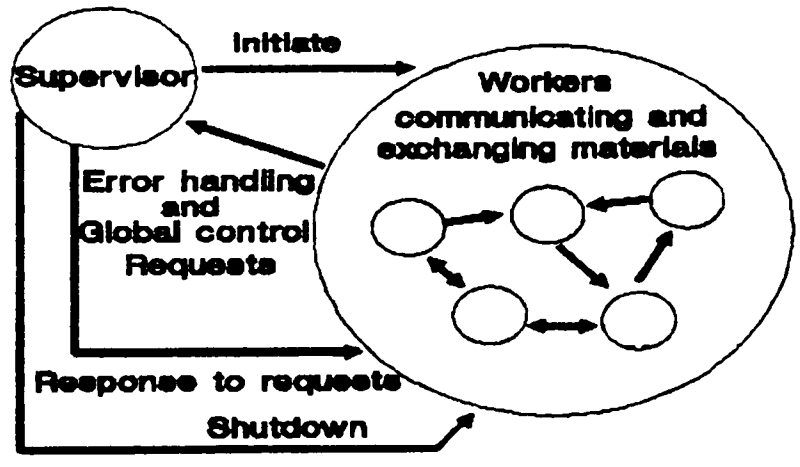


Figure 3.6. System with a supervisor.

To initialize the system, the supervisor object sends messages to all objects. Each object should have a specific method which performs the initialization steps if needed. The initialization methods may be inherited from the superclass object, instead of being redefined for each object. The message from the supervisor object carries this method, and the type of this message is an operation invocation. Upon receiving the initialization request messages from the supervisor object, all objects send confirming messages or error handling messages (if any errors are found) back to the supervisor object. When the supervisor object receives the confirming messages from all objects in the system, the system is ready for operation.

How to Handle Message Traffic

Figure 3.7 shows how the incoming messages are processed in the object. When a message is received, it is handled by one of the utility routines in the destination object. That is, from the programmer's point of view, the received message is handled by one of the subprograms in the main body of the program which implements the object. When the message is sent to the message queue, the information on the received message is

also sent to the specific object whose task is to keep track of which messages have been received and processes. We will call this object a "log object." Depending on software implementation, the log object can be incorporated into the supervisor object, or can exist as a separate object. For simplicity, Figure 3.7 shows only those elements which are directly involved in message processing.

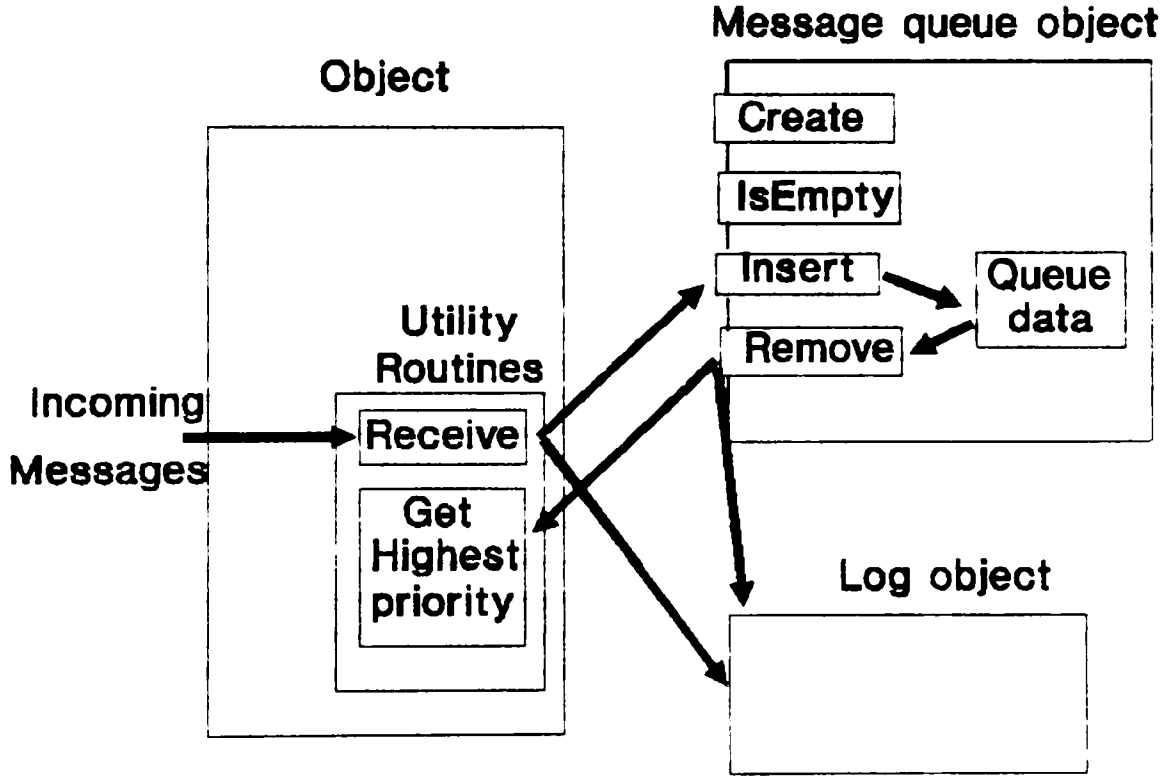


Figure 3.7. Message processing.

In a message-driven environment with heavy message traffic, each object may receive far more messages than it actually processes at one time. Therefore, all the incoming messages are placed into the message queue and dispatched according to its priority. How to set up and deal with priority values is addressed below.

How to Set Up and Deal With Priority Values

The priority value is denoted as any integer, with the higher the integer, the greater the priority. When an object sends a message, it decides the urgency of the message and attaches the appropriate priority value to the message. Therefore, objects should be implemented, so that whenever a message is sent, an appropriate code to determine the priority value, or a hard-coded value of priority, should be included. The error handling messages and the messages from the supervisor object may have the higher priority values, so that these messages would override the normal sequence of processing. Most operation invocation messages may have priority values which are lower than for error handling messages and messages from the supervisor object, but higher than the notification messages.

The Message queue object is implemented as a priority queue, so that the message with the highest priority value is easily retrieved. If more than one message have the same priority value at a specific moment, any message which is retrieved first from the message queue, depending on the implementation of the priority queue, will be taken care of first.

Error Handling

It is very important to handle errors and failures effectively in real-time systems. Each object needs to include the code to detect any possible problems. This error handling procedure will be included in the utility routines. Most common errors in the message passing activities are (a) the requested method is not found in the destination object, or (b) the type and numbers of parameters are not matched with those defined in the destination object. In such cases, the destination object which received the incorrect messages will send an error handling message back to the source object. Also, when the requested operation cannot be finished within the specified time, the utility routine sends an error handling message to the sender object.

3.4 Representation of the OOART

Requirements have traditionally been represented in an English textual form. However, a graphical representation of the system architecture can be more easily understood and analyzed without ambiguity than a textual representation. Therefore, the following diagrams and dictionaries will be employed to represent the OOART: the abstraction layer diagram, the inheritance diagram, the message passing diagram, the object dictionary, and the message dictionary. The purposes, notations, and basic workings of the above diagrams and dictionaries are explained in this section.

How to Represent the Objects

Object. An object is the basic unit of the object-oriented systems. Each object is depicted as a box with its identifier at the top in a closed slot, its method names inside the protruding rectangles, and its attributes inside the ellipses which reside in the second slot of the object body. As shown in Figure 3.8, only the names of its methods and attributes are shown in an object representation. This convention supports the concepts of encapsulation and information hiding. All operations in the methods and utility routines in an

object are not visible outside the object. Also, the message queue object is not explicitly displayed even though each object should have a message queue object connected to it.

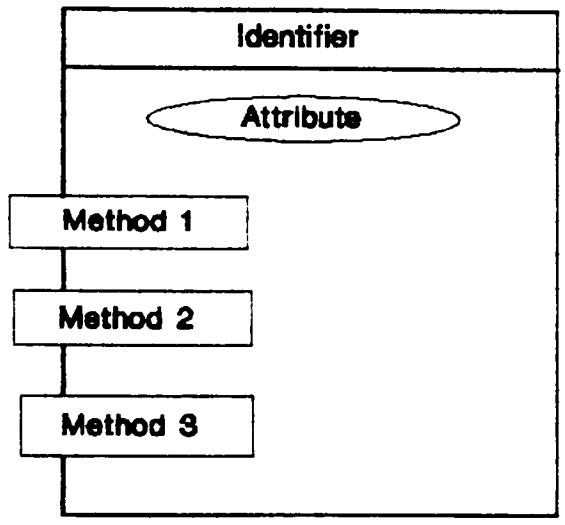


Figure 3.8. Notation for an object.

Object dictionary. Further details of the objects are maintained in an object dictionary as shown in Figure 3.9. This dictionary contains the following information:

1. Object identifier to keep track of the identifiers defined and to avoid the duplicate names.

2. Methods and parameters to show the available methods and any parameters required by the specified methods. If the methods are inherited from another object, it will be also indicated with a flag "I:." If

an object inherits the methods and adds new methods to them, this newly added methods will be indicated with a flag "A:." Also, any methods which are overridden will be noted with a flag "X:."

3. Attributes to show the specified attributes for an object. Same as the methods, the inherited attributes will be flagged as "I:." Newly added attributes and attributes override will be distinguished with a flag "A:" or "X:," respectively.

4. Superclass to indicate the object from which the methods or attributes are inherited.

Object identifier	Methods if needed	Parameters	Attributes	Super class
Object1	Method1	P1	Attr1	--
	Method2	P2	Attr2	
	Method3		Attr3	
Object2	I: Method1	P1	X: Attr1	Object1
	I: Method2	P2	I: Attr2	
	I: Method3		I: Attr3	
			A: Attr4	
Object3	I: Method1	P1	I: Attr1	Object1
	X: Method2		I: Attr2	
	X: Method3		I: Attr3	
	A: Method4			

Figure 3.9. Object dictionary.

For example, an object dictionary shown in Figure 3.9 indicates that the system contains three objects and Object2 and Object3 are inherited from Object1. Object2 inherits all three methods and two attributes (Attr2 and Attr3) from Object1, hides one attribute (Attr1), and newly adds an attribute (Attr4.) Information on inheritance also can be depicted in an inheritance diagram, which will be discussed later.

How to Represent the Relationships Between Objects

The relationships between objects are depicted in the abstraction layer and inheritance diagrams. Information on inheritance is also maintained in the object dictionary, as shown in Figure 3.9, and the communication channels between objects are represented in the message passing diagram. The message passing diagram will show the availability of other objects to any specific object in terms of communication.

Abstraction layer diagram. Large object-oriented systems are built in layers of abstractions. Putting every object of an entire system in one diagram would not only clutter the diagram, but also make the diagram difficult to understand. Therefore, to reduce the amount of complexity that must be comprehended at one time, and

to cluster the objects into the meaningful groups, different layers of abstraction are used. Each layer of such a collection of objects corresponds to a subsystem. Notations for an abstraction layer diagram are shown in Figure 3.10. Each layer is depicted as a (large) rectangle, with its layer number shown in the slot at the upper left corner of the box. Each layer also can have its own name and have it displayed along with the layer number. Layer 0 is like a context diagram in the structured analysis; it shows the hierarchy of the entire system. It can contain any of the lower level layers and objects, which are represented as rectangles and circles, respectively. To simplify the diagram, only the identifiers of the objects are shown in abstraction layer diagram. The abstraction layer diagram can show the hierarchy of layers and objects, or just put them in parallel to show that they are structurally flat. Any layers of lower level which are shown in the upper layer diagram should be refined in a separate diagram.

For example, Figure 3.10 (a) shows that the system contains three different layers which are structurally flat. Layer 1 contains a hierarchical collection of objects, where an object named O1 is a parent object and the objects O2, O3, and a set of objects which is grouped

as layer2 are descendants of O1. Layer 2 is further refined in Figure 3.10 (c).

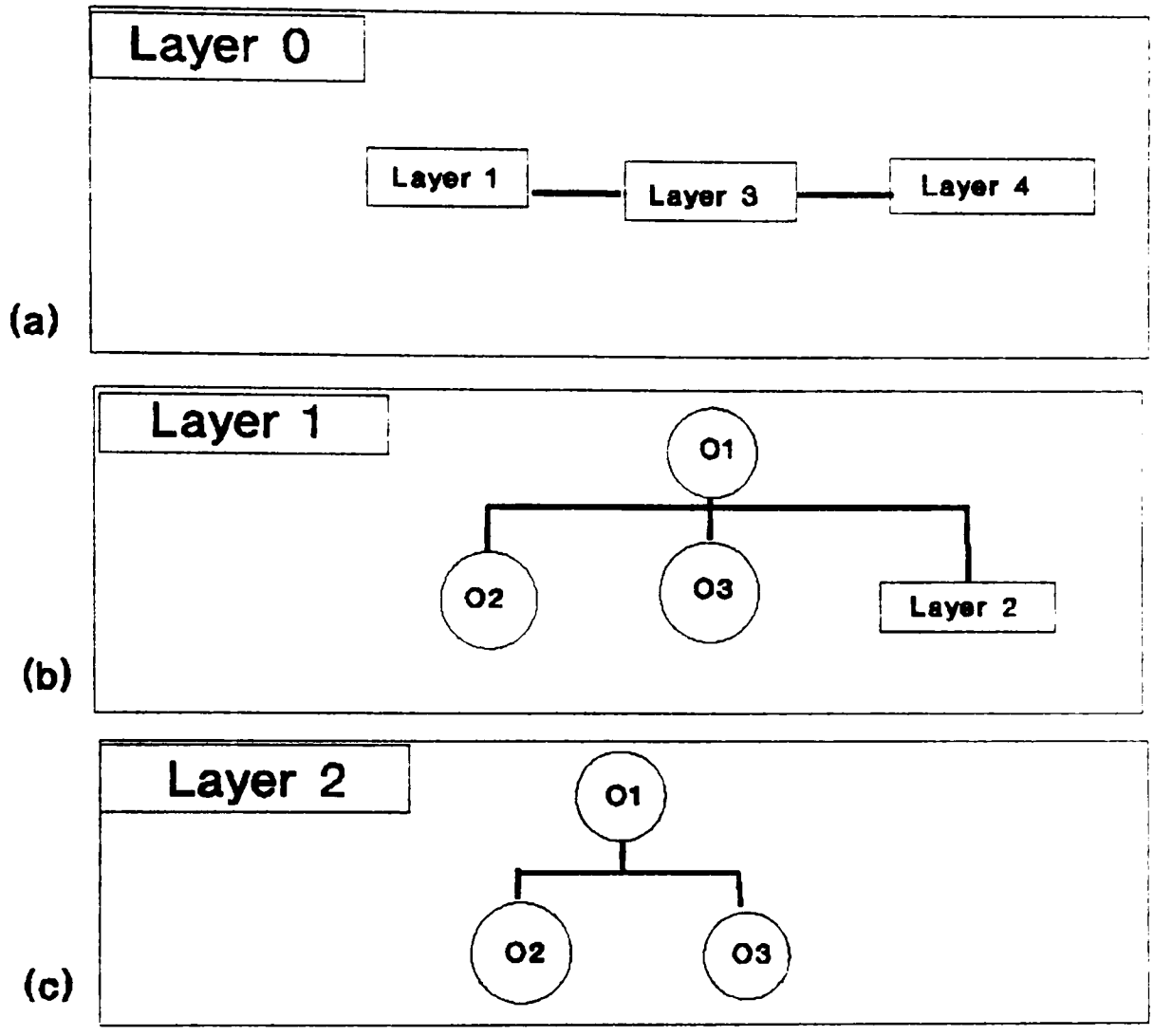


Figure 3.10. Abstraction layer diagrams.

Inheritance diagram. Inheritance is one of the important characteristics of object-oriented systems. The inheritance diagram, which is used frequently in the literature, will be employed in the OOART to support the concept of inheritance [53, 54]. Figure 3.11 shows

::

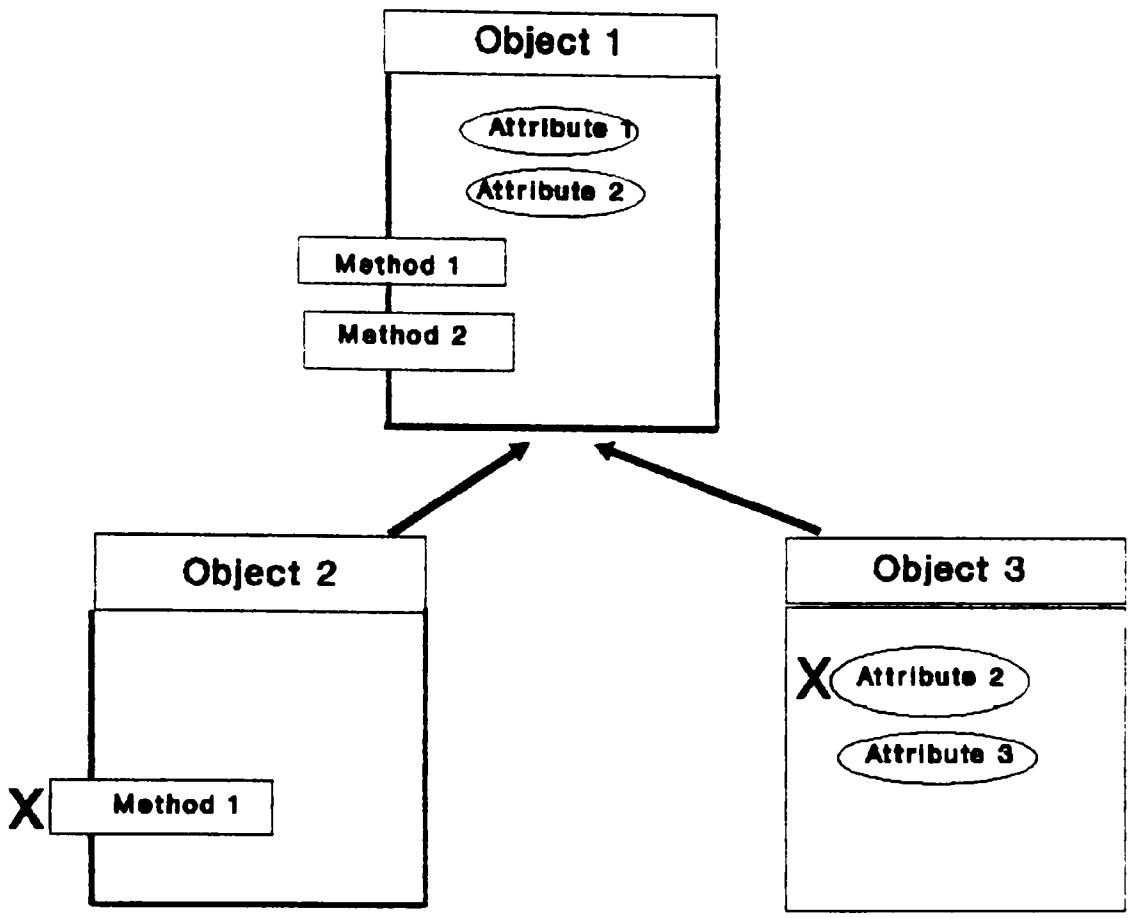


Figure 3.11. Inheritance diagram.

an inheritance diagram in which Object2 and Object3 are inherited from Object1. The directed arc shows the inheritance relationship between two objects. Information on which methods and attributes are inherited, deleted (override), or newly added are also shown in this diagram. While an object dictionary shows every inherited methods and attributes with a flag "I:," they are not repeated in an inheritance diagram. Only the changes (newly added or deleted) are indicated explicitly. Methods and attributes overrides are shown

with the "X" notation in front of them. In Figure 3.11, Object2 overrides Method1, which was inherited from Object1. So, Object2 contains only one method (Method2) and two inherited attributes (Attribute1 and Attribute2). Object3 overrides one inherited attribute (Attribute2) and adds one new attribute (Attribute3), while keeping all inherited methods.

How to Represent the Message Passing

Information on message passing between objects is depicted in a message passing diagram, and the details on messages are maintained in a message dictionary.

Message passing diagram. A message passing diagram shows the channels of messages between objects. Figure 3.12 shows the notations for the message passing diagrams. A directed arc connecting two objects denotes the potential to communicate between the two indicated objects, and the label on the arc denotes the message number by which the details of this message will be placed in a message dictionary. The source object which sends the indicated message has a dot beside each of its methods, and a directed arc starts from any specific dot showing which method is sending the message. If there is no method along with the specified dot and the arc starts

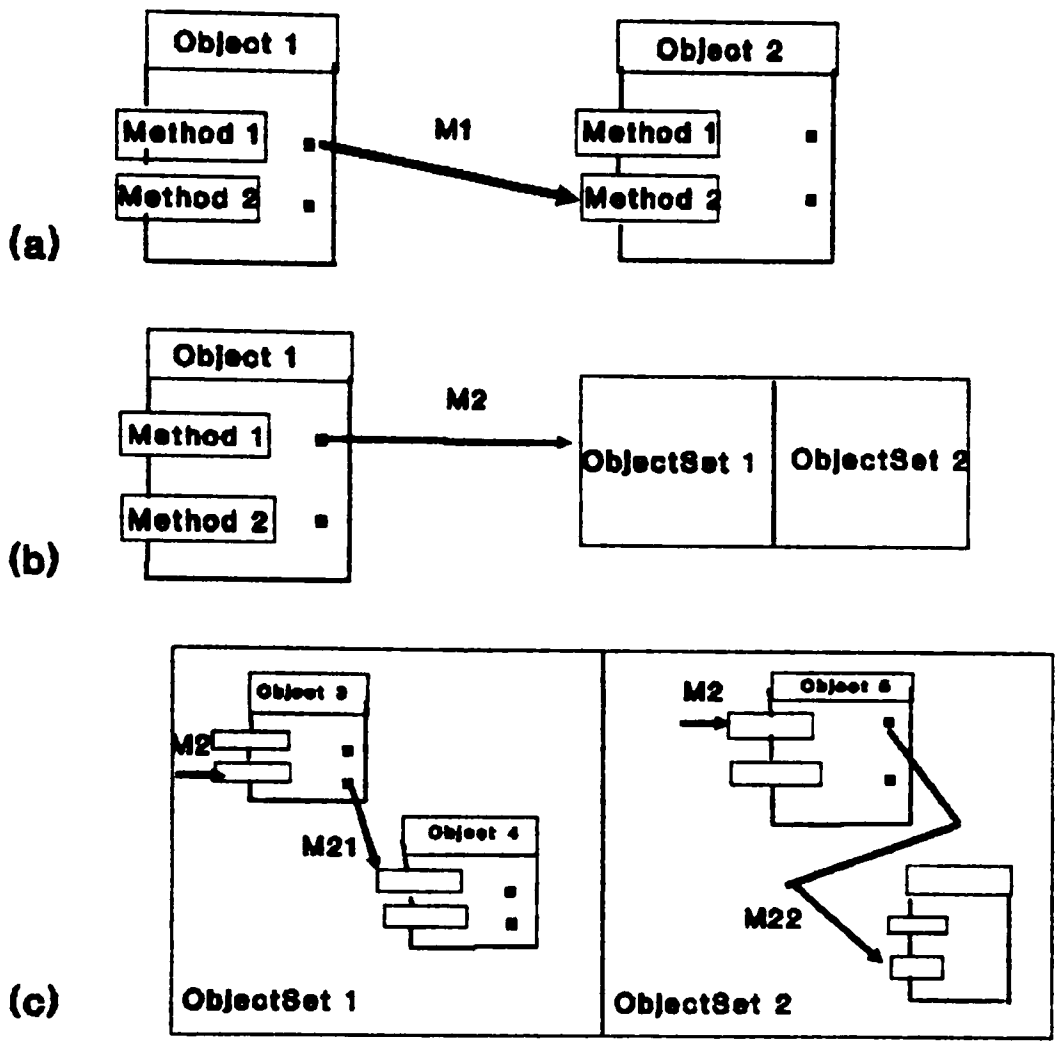


Figure 3.12. Message passing diagrams.

from that dot, it means that any one of the utility routines sends the specified message.

Also, the directed arc points to any specific method to indicate the requested method. In a message passing diagram, only operation invocation messages are shown. The error handling and notification message are not shown explicitly.

Figure 3.12 (a) shows that Method1 of Object1 sends a message to Object2 to request the execution of Method2. To support the concurrency of the system, a set of objects is grouped together to form "ObjectSet." As shown in Figure 3.12 (b), Object1 can send the messages either to ObjectSet1 or ObjectSet2. Then, the further message passing activities in ObjectSet1 and ObjectSet2 are refined in a separate diagram as shown in Figure 3.12 (c). On receipt of the specified message from Object1, Object3 in ObjectSet1 activates the indicated method, which in turn sends another message to Object4. In the other hand, the ObjectSet2 shows how messages may be passed if Object5 receives a message from Object1. The semantics of this refinement is regarded as "or" function. The "and" function in which two sets of objects are sending messages concurrently will be denoted by splitting a box using dashed lines instead of solid lines. This diagram is based on the statecharts which are the extensions to Finite State Machines (FSM), but modified to incorporate the concepts of object [10].

Message dictionary. A message dictionary keeps track of all messages which are shown in the message passing diagrams. The following information will be placed in this dictionary: message number, source

identifier, destination identifier, message contents, parameters, priority value, and time constraints. Figure 3.13 shows how the message dictionary will look like.

Msg Number	Source Id.	Dest. Id.	Msg Contents	Param.	Priority Value	Time Constraints
1	O1	O2	Method1		7	
2	O1	O3	Method2		6	
.						
.						
.						

Figure 3.13. Message dictionary.

CHAPTER IV

APPLIED RESULTS

A multi-process single wafer cleaning system (the FSI project) has been used as a case study for the OOART. This system is used for distributed real-time control of semiconductor process equipment. The Center for Research in Robotics and High Technology at Texas Tech University in Lubbock, Texas has utilized the object-oriented approach to implement cluster control of advanced wet and dry chemical cleaning of silicon wafers [30]. Part of the research descriptions from [30] has been reproduced in Section 1 to show what the system is supposed to do and to review how it is currently analyzed and handled in terms of the message passing techniques. Then, Section 2 describes the application of the OOART to this real-time system and shows how the system is graphically represented using the OOART notations.

4.1 Case Study Review: Message Passing Techniques for the Current FSI Project

Single Wafer Cleaning System

Wafer cleaning is required numerous times in the production of integrated circuits, and becomes

increasingly critical as device geometries grow smaller. The methods for cleaning wafers include dry clean, wet clean, thermal and ultrasonic technologies. Within the cluster tool, there will be a need to carry out metrology as well as wafer transport including one or more load and unload stations. The cluster controller for multi-process cleaning of silicon wafers implements decentralized process control, which integrates the process steps into a single unified system. The control systems are based on multiple computers which do not share primary memory and which communicate by means of a LAN. A centralized computer provides global coordination of all resources within the cluster; inter-node communication within the cluster must be managed by a cluster controller. Using the object-oriented approach, the system is implemented as a set of interacting objects [30].

Message Passing Relationships

Communication between objects is done by message passing. Figure 4.1 shows some of the likely message passing relationships that would exist when the objects are integrated to implement a cluster controller. The

following paragraphs describe a set of operations that act on objects via passing messages.

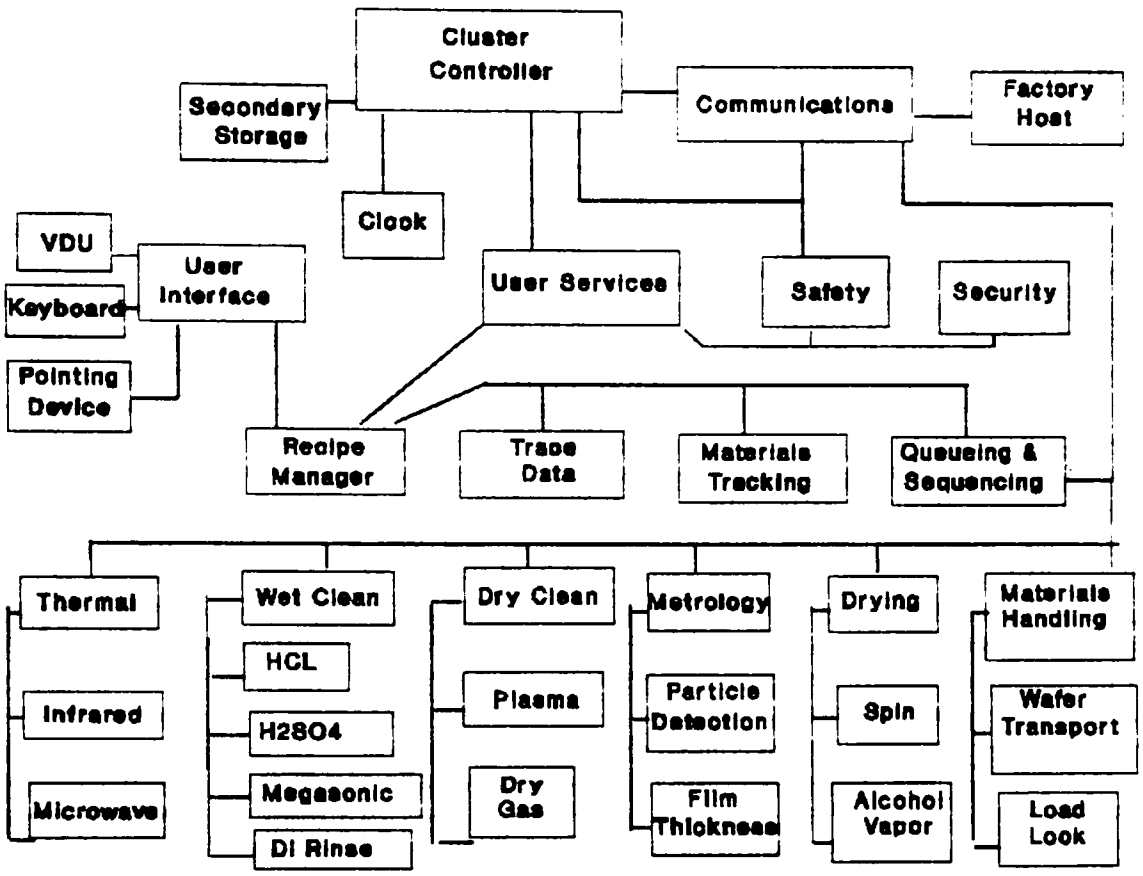


Figure 4.1. Network diagram of objects passing messages.

Initialization. The Cluster Controller initializes the system by sending messages to all objects asking them to initialize themselves. All objects send confirming messages to Safety informing it that all objects have been successfully initialized. Safety sends a message to the Cluster Controller informing it that the system is ready for operation.

Operation. The cluster controller sends a message to User Services asking it to open communications with the User Interface. User Services sends a message to the User Interface asking it to display a login prompt and respond with the operator's identification and password. Upon receiving the operator's identification and password, a message containing this information is sent to Security for verification. Security responds appropriately sending a message to User Services.

If the operator is granted access, User Services sends a message to the User Interface to prompt the operator to place a cassette of wafers in the Load Lock. User Services also sends a message to Materials Handling to report when the cassette is loaded and ready for processing. As soon as wafers are loaded and available User Services sends a message to the Recipe Manager to assume control.

The Recipe Manager sends a message to the User Interface to display the recipe control menu and awaits operator input. The Recipe Manager also sends message to Trace Data and Materials Tracking which in turn send messages to Secondary Storage to open the files that will store wafer processing data as it occurs. When the operator selects a recipe, the Recipe Manager interprets

the recipe and sends a message stream to Queuing and Sequencing.

Queuing and Sequencing queries Safety and then sends prioritized messages to Materials Handling to load the next wafer in the designated Process Module. As each message is acknowledged, Queuing and Sequencing continues to send messages to Materials Handling and each Process Module to process the wafer through the specified steps.

Some actions are conditional upon message being received from the Process Modules. The messages may contain data for Trace Data or may be sent to a Process Model which computes a modified process parameter for a subsequent step. The Process Model sends a message to Queuing and Sequencing to update one or more process parameters.

As each step of a recipe is completed, Materials Tracking receives messages about where wafers reside in the system and sends messages to Safety so that shared system resources can be locked and unlocked. If multiple process modules are available, several wafers can be processed concurrently. The process recipe actually contains not only the process programs for each module but also the routing and priority information for materials handling within the cluster controller.

Eventually all the wafers in a cassette will be processed and another set of wafers will be loaded.

Shutdown. If no more processing is to be done or if the machine is to be shut down, the operator enters a reset or escape sequence at the User Interface. The User Interface sends a message to User Services requesting a reset. User Services sends messages to Safety, Materials Tracking, Trace Data and Queueing to perform an orderly shut down of processing. When the reset is complete control returns to the Cluster Controller and the system is re-initialized. At this point the system can be left in a quiescent state or new processing initiated [30].

4.2 Case Study: Apply the OOART to a Single Wafer Cleaning System

As a case study of this research, the OOART has been applied to the single wafer cleaning system as described above. This section shows the application of the four steps in the OOART, along with the resulting diagrams and dictionaries which would help to represent the system.

Recognize the Preliminary Objects

The first step in the OOART is to look at the problem space and recognize the major entities that should play their roles and have attributes in a model of reality. In this case study, the hardware and software objects which have been developed in the Single Wafer Cleaning System of the FSI project [30] are used as starting blocks. However, the following changes have been made to these objects in order to apply the OOART more appropriately:

1. Instead of having a Cluster Controller, which manages inter-node communication of the objects in the system, a Supervisor object is used. This Supervisor object performs the initialization and shutdown of the system, and gives global coordination to the objects. However, unlike the Cluster Controller, not all messages must pass through the Supervisor object. The Supervisor object does not interfere with message passing of other objects of the system unless an error occurs, or unless an object sends a request for any global control to the supervisor object. Even though most of the errors will be handled in the utility routines in each object, any errors which might need a global coordination would be sent to the supervisor object to be handled.

2. The granularity of objects has been modified, so that the system can be effectively represented in the OOART. For example, the Wet Clean object incorporates HCL (Hydro chloride), H2SO4 (Sulfuric acid), Megasonic, and DI Rinse as its methods instead of having them as separate descendant objects.

3. The Recipe Manager object performs the tasks which have been done by the Queueing & Sequencing object. Since each object has its own message queue, the Recipe Manager object takes the incoming messages (operator's selection of recipe) and dispatches them to the Material Handling object and each Process Modules according to its priority values.

Identify the Attributes and Operations

The second step is to recognize the attributes and operations for the derived objects. Figure 4.2 shows an object dictionary which contains this information on each object of the single wafer cleaning system. The object dictionary also shows any parameters required by the specified methods, and the superclass from which the methods or attributes are inherited.

Object Identifier	Method	Parameters	Attributes	Super Class
Supervisor	1.Initialize 2.Start_operation 3.Shut_down 4.Global_control		System_mode	
User Interface	1.Initialize 2.Receive_input 3.Display_output	Input _contents Output _contents	Input_format Output_format	
Input Device	1.I:Initialize 2.I:Receive_input X:Display_output		I:Input _format X:Output _format	User Interface
Output Device	1.I:Initialize 2.I:Display_output X:Receive_input		I:Output _format X:Input _format	User Interface
Process Modules	1.Initialize 2.Select_module	Selection	Process _status	
Dry Clean	1.I:Initialize 2.A:Plasma 3.A:Dry_gas X:Select_module	Selection Selection	I:Process _status	Process Modules
Wet Clean	1.I:Initialize 2.A:HCL 3.A:H2SO4 4.A:Megasonic 5.A:DI_rinse X:Select_module		I:Process _status	Process Modules
Drying	1.I:Initialize 2.A:Spin 3.A:Alcohol_vapor X:Select_module		I:Process _status	Process Modules

Figure 4.2. Object dictionary for a single wafer cleaning system.

Object Identifier	Method	Parameters	Attributes	Super Class
Thermal	1.I:Initialize 2.A:Infrared 3.A:Microwave X:Select_module		I:Process_status	Process Modules
Metrology	1.I:Initialize 2.A:Particle_detection 3.A:Film_thickness X:Select_module		I:Process_status	Process Modules
Plasma	1.I:Initialize 2.A:Microwave 3.A:RF		I:Process_status	Dry Clean
Dry Gas	1.I:Initialize 2.A:Ozone 3.A:Anhydrous_HF		I:Process_status	Dry Clean
Materials Handling	1.Initialize 2.Wafer_transport 3.Load_Lock	 Wafer_num Wafer_loc Wafer_num Wafer_loc	Lock/unlock_status	
Communi-cation	1.Initialize 2.LAN 3.Factory_host			
Clock	1.Send_current_time		Time_data	
Log	1.Initialize 2.Receive_log_data 3.Output_log_data	 Log_data	Log_data	
User Services	1.Initialize 2.Open_communication 3.Start_processing 4.Request_reset			

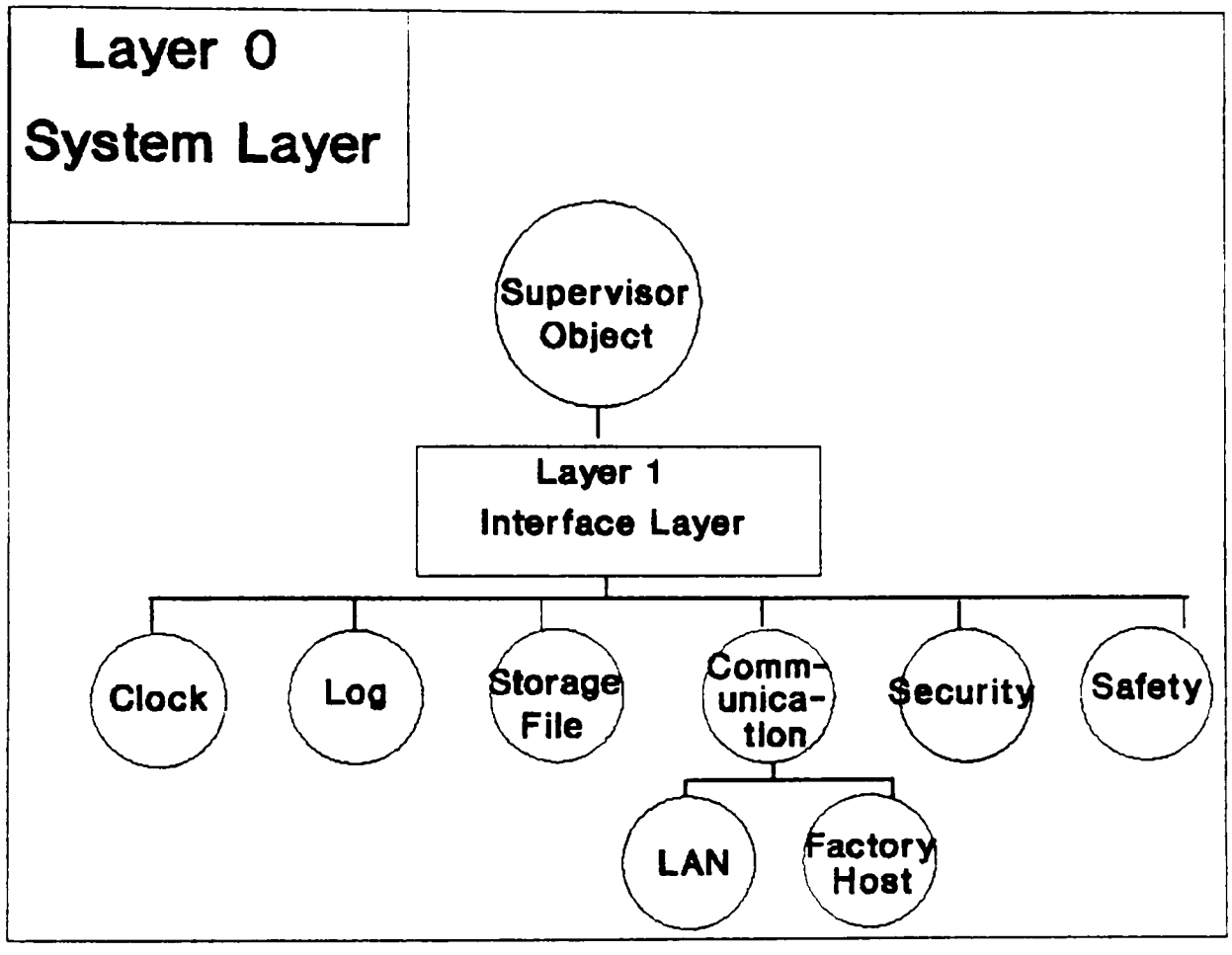
Figure 4.2. Continued.

Object Identifier	Method	Parameters	Attributes	Super Class
Recipe Manager	1.Initialize		Recipe_info	
	2.Receive_control			
	3.Receive_recipe	Recipe		
	4.Log_data	Log_data		
	5.Request_process_modules	Module Wafer_num		
	6.Request_materials_handling	Module Wafer_num		
	7. Shut_down			
Trace Data	1.Initialize		Wafer_process_data	
	2.Store_data	Process_data		
	3.Retrieve_data			
	4.Shut_down			
Materials Tracking	1.Initialize		Materials_info	
	2.Receive_wafer_location	Wafer_num Wafer_loc		
	3.Shut_down			
Security	1.Initialize		Security_info	
	2.Verify_access	Id/password		
Safety	1.Initialize		resource_info	
	2.Lock_resource	Resource		
	3.Unlock_resource	Resource		
	4.Shut_down			
Storage File	1.Initialize		Wafer_process_data	
	2.Store_data	Process_data		
	3.Retrieve_data		Wafer_process_data	

Figure 4.2. Continued.

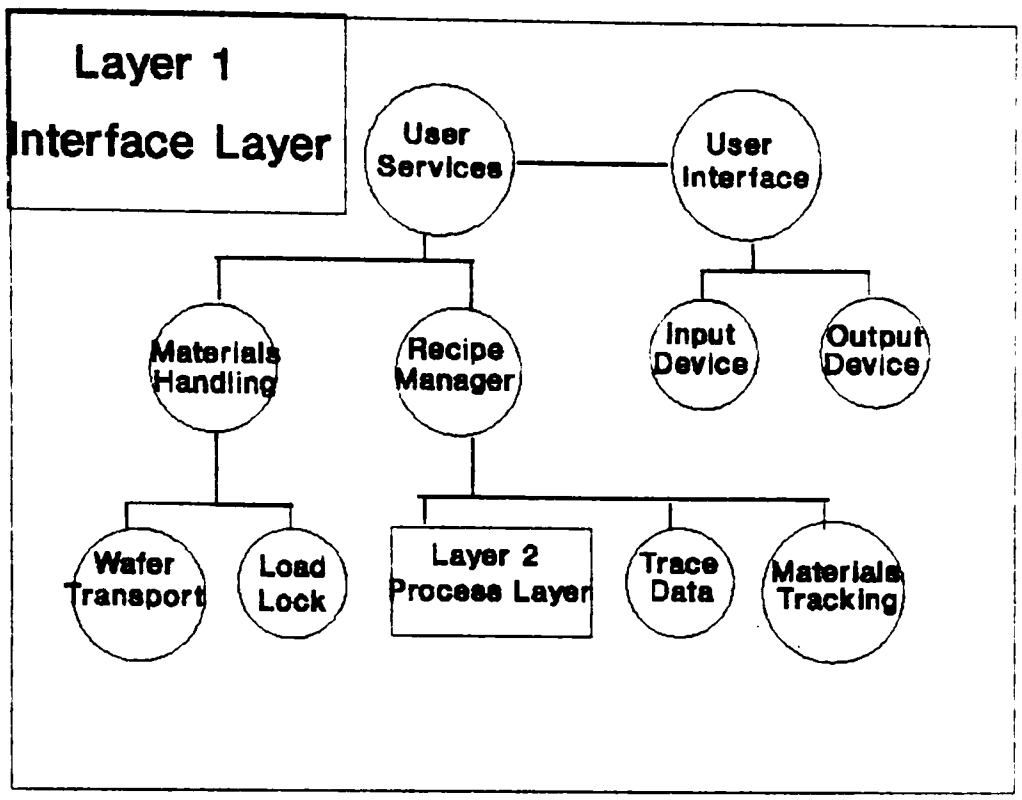
Build the Layers of Abstraction

The next step is to build the different layers of abstraction. For the single wafer cleaning system, three different layers are used: System Layer, Interface Layer, and Process Layer. They are shown in Figure 4.3. The abstraction layer diagrams show the hierarchy of layers and objects, but the detailed information on inheritance is shown in the inheritance diagrams in Figure 4.4.

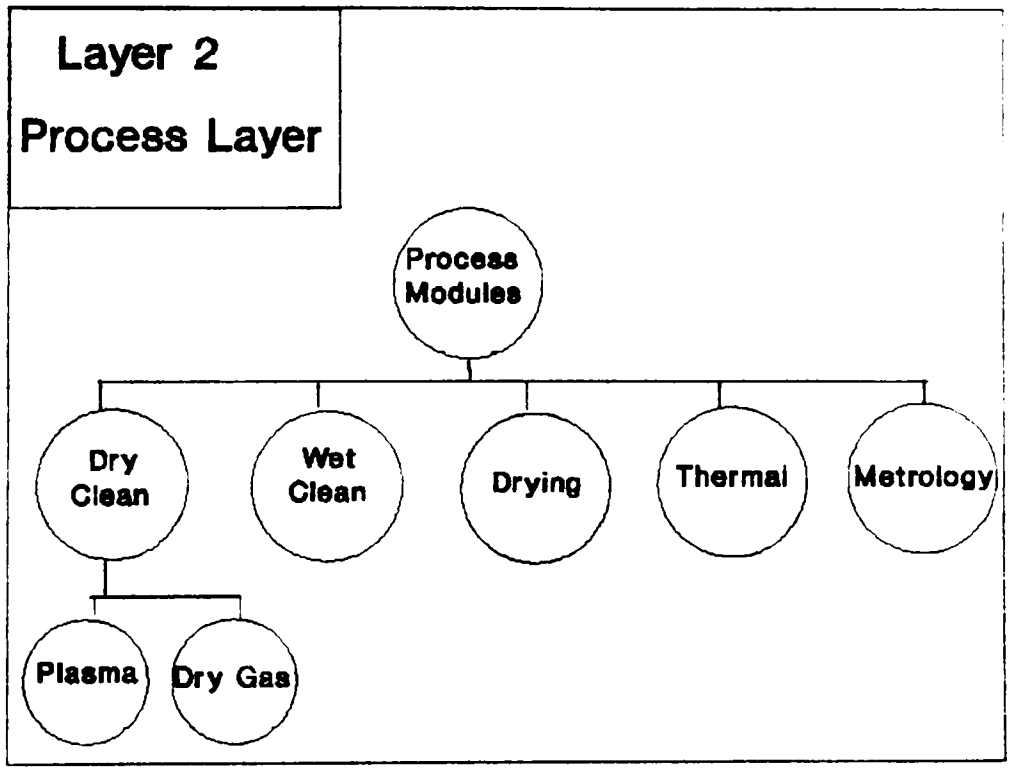


(a) Layer 0: System Layer.

Figure 4.3. Abstraction layer diagrams for a single wafer cleaning system.

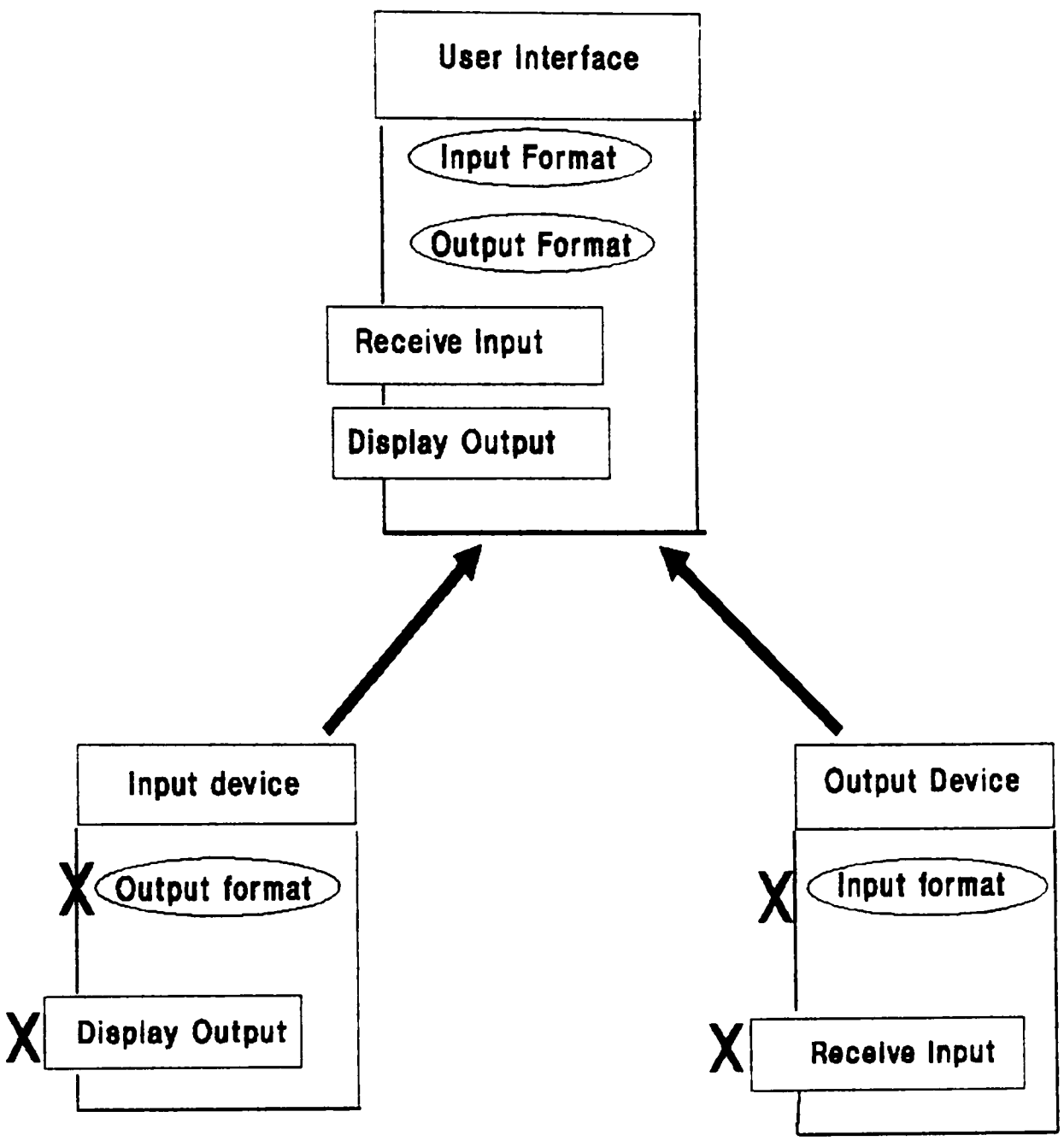


(b) Layer 1: Interface Layer.



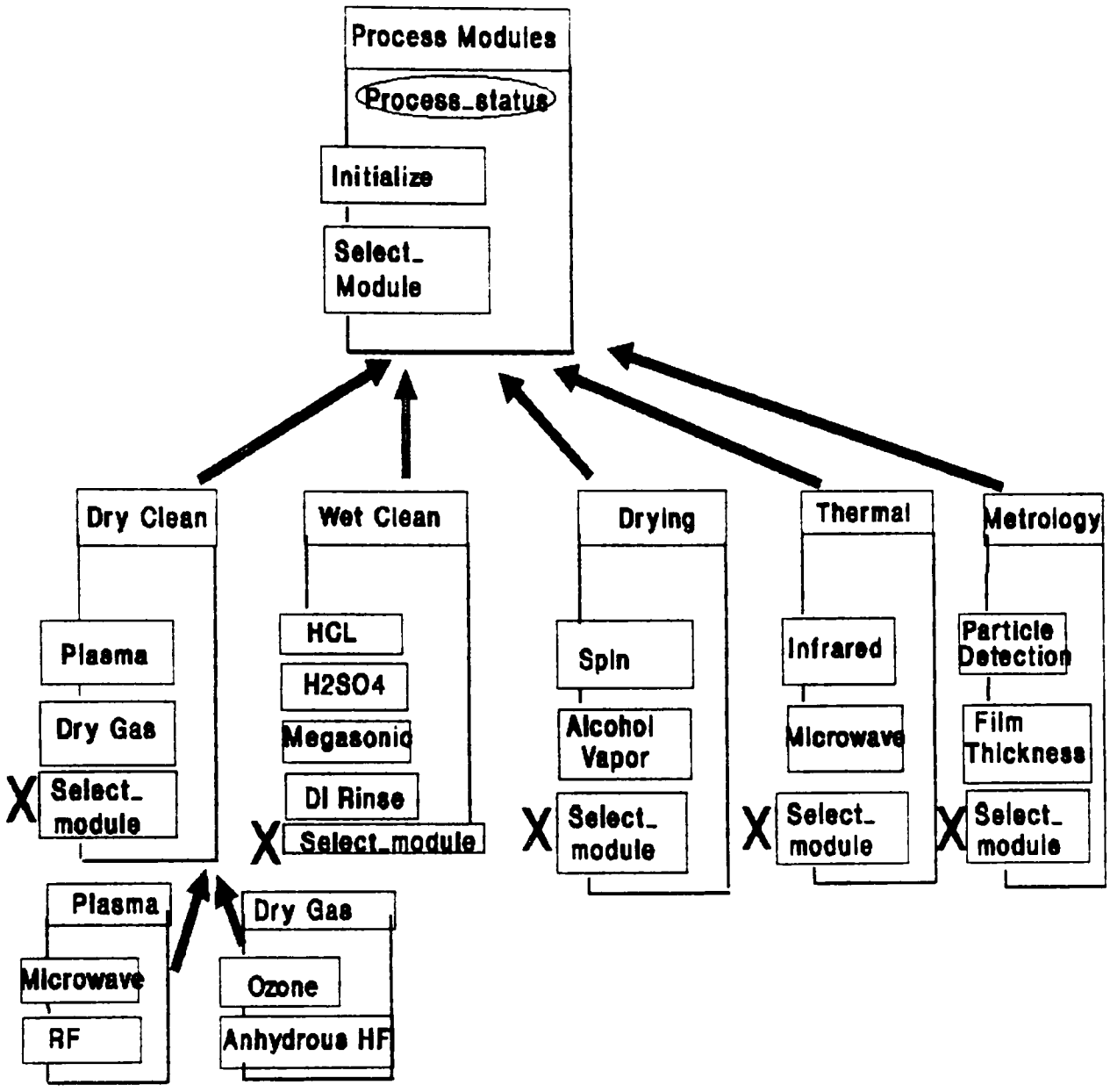
(c) Layer 2: Process Layer

Figure 4.3. Continued.



(a) Inheritance diagram for the User Interface.

Figure 4.4. Inheritance diagrams for a single wafer cleaning system.



(b) Inheritance diagram for the Process Modules.

Figure 4.4. Continued.

Establish the Interfaces

As the final step, message passing between objects is defined and represented in the message passing diagrams. Figure 4.5 shows the message passing diagrams for initialization of the system. As shown in Figure 4.5(a), the supervisor object sends messages to all objects asking them initialize themselves. The objectSet 1 (All_objects) represents all objects which will receive the initialization messages and Figure 4.5(b) shows the further refinement of this ObjectSet. Dashed lines are used in splitting a box to indicate that all of the objects in ObjectSet 1 should receive the initialization messages (as "and" function,) and upon receiving the messages each object concurrently performs the predefined initialization steps, and sends confirming messages to the supervisor object. After receiving the confirming messages from all objects, the supervisor object changes its system_mode from initialize to operation and the system is ready for operation.

Figures 4.6 and 4.7 show the potential message passing activities for operation and shutdown, respectively. Figure 4.6(a) shows the three major objectsets that the user services interacts with for operation: Start_mode, Load_wafer_mode, and Process_mode.

Figures 4.6(b) through 4.6(n) show the further refinements on each objectsets.

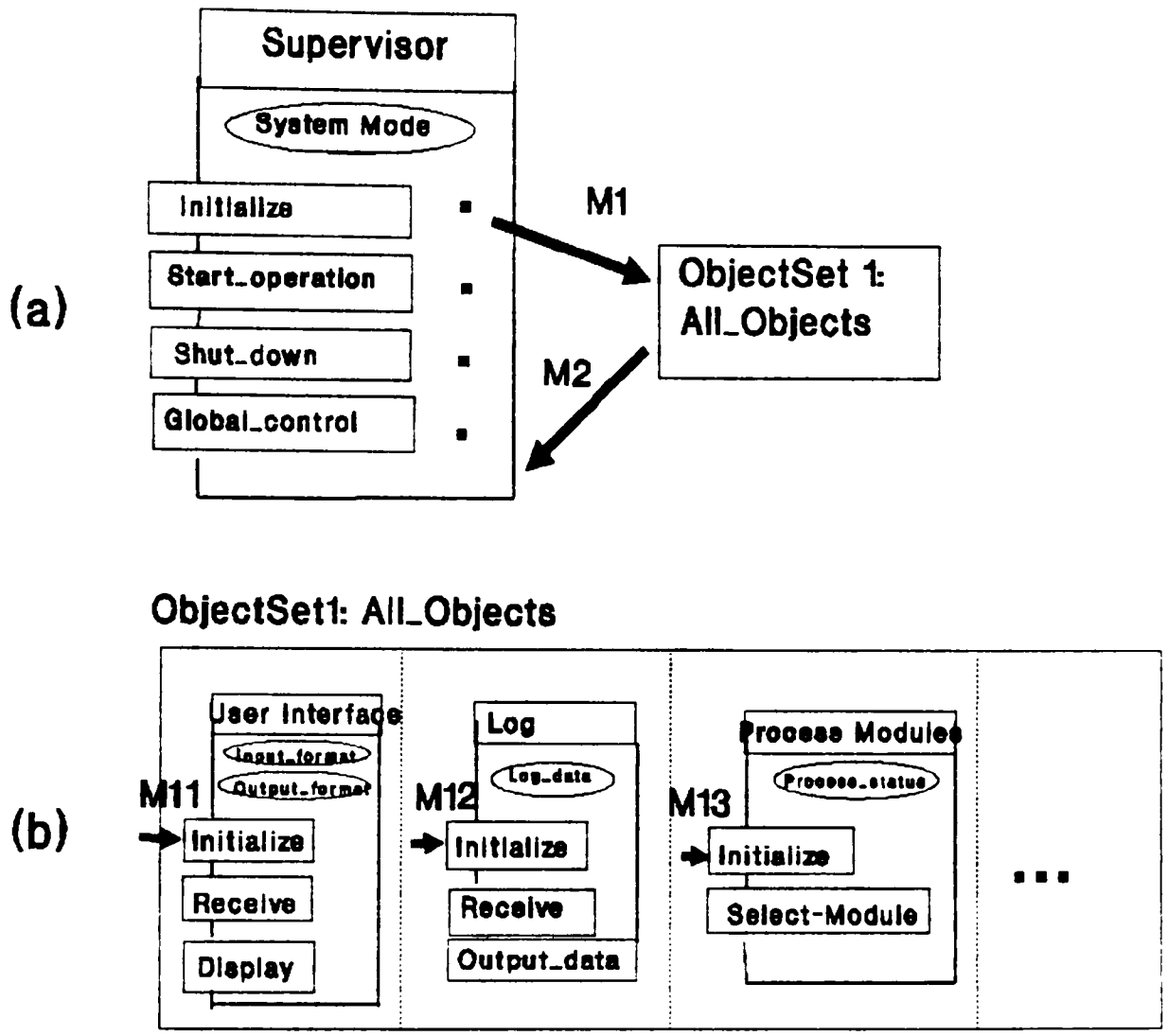
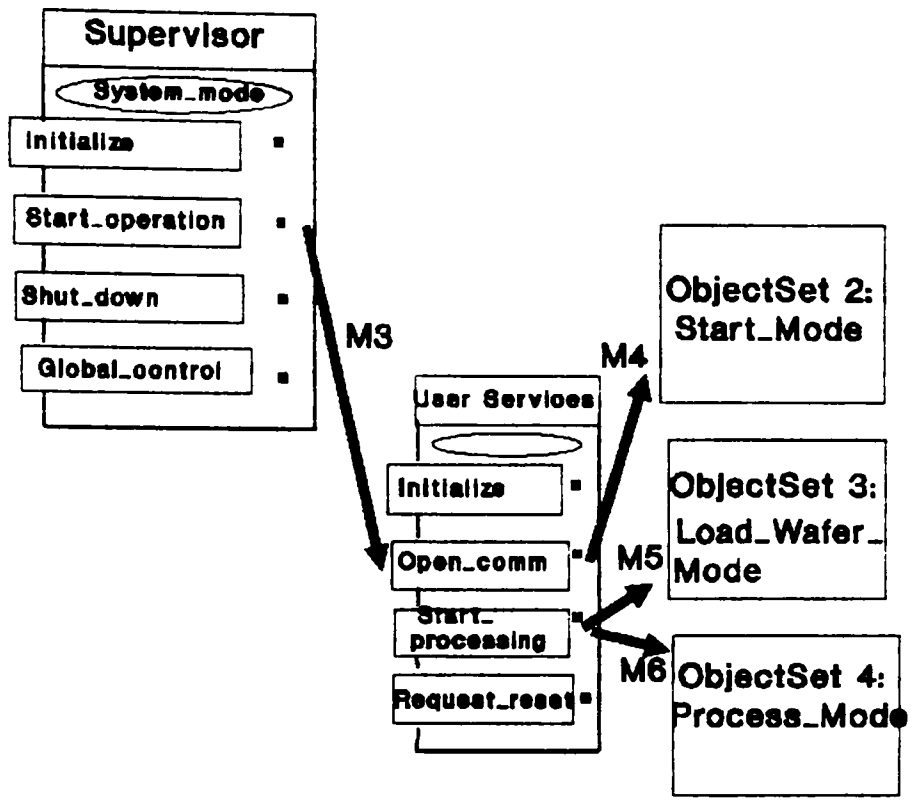


Figure 4.5. Message passing diagrams for initialization of the single wafer cleaning system.

(a)



(b)

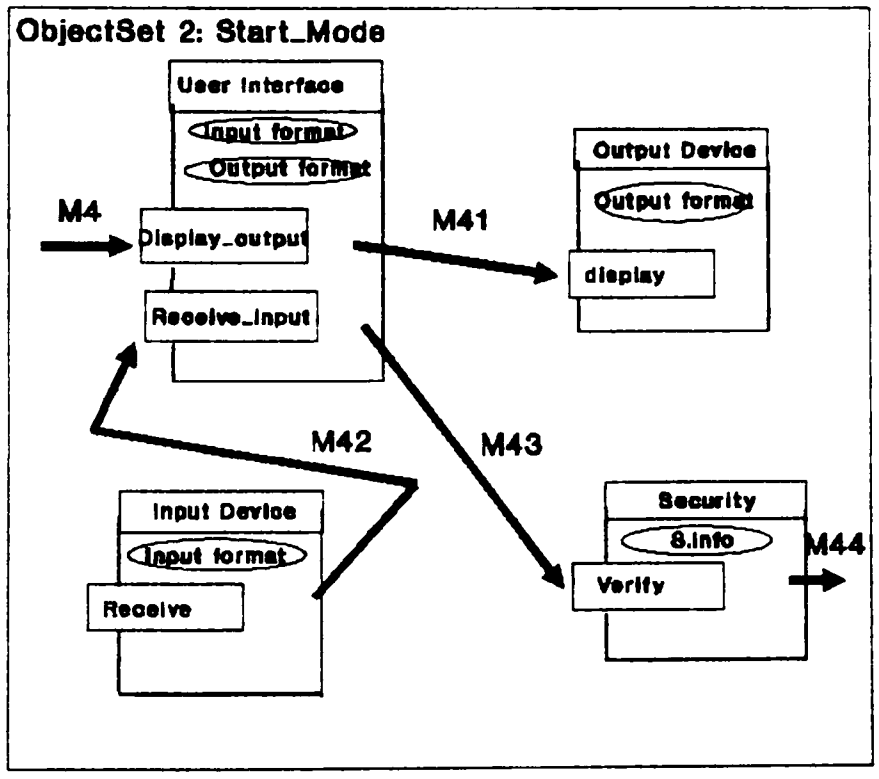
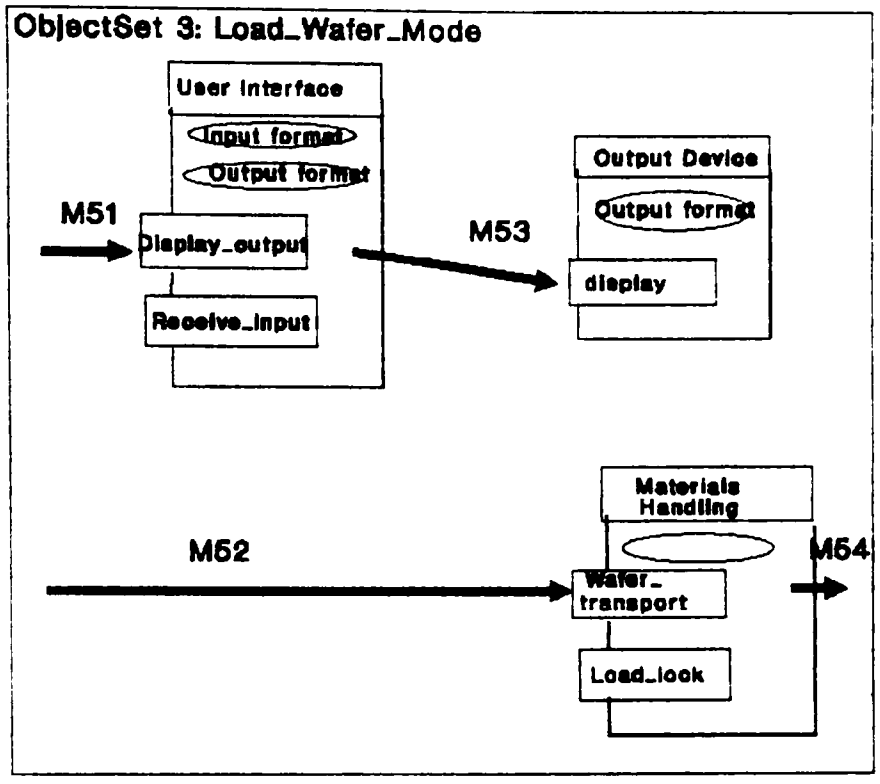


Figure 4.6. Message passing diagrams for operation of the single wafer cleaning system.

(c)



(d)

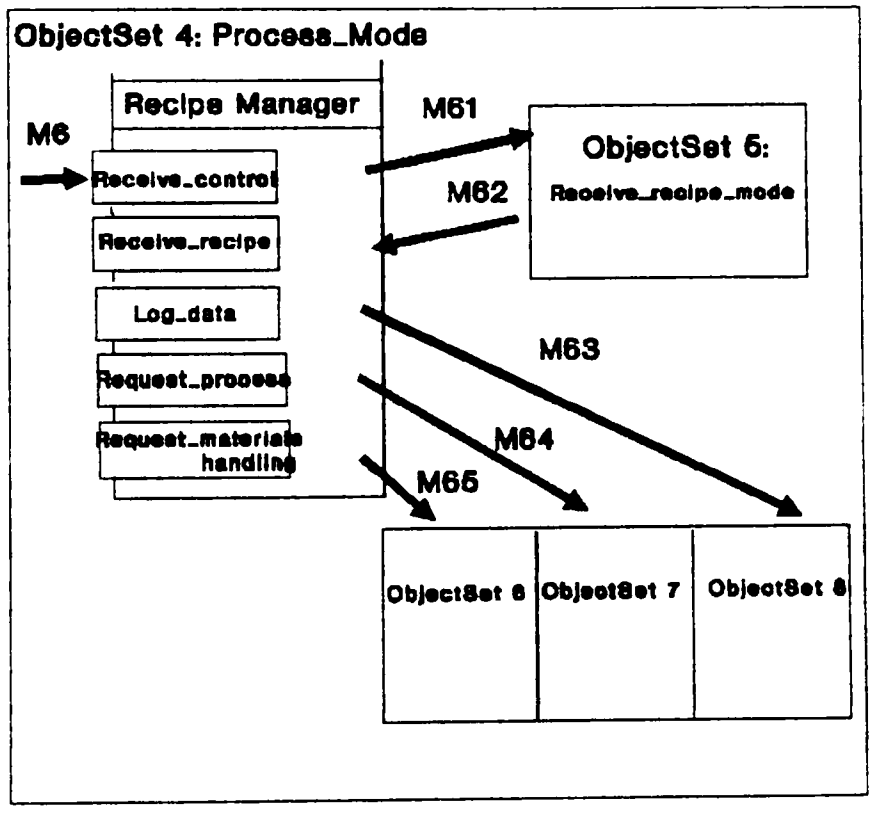
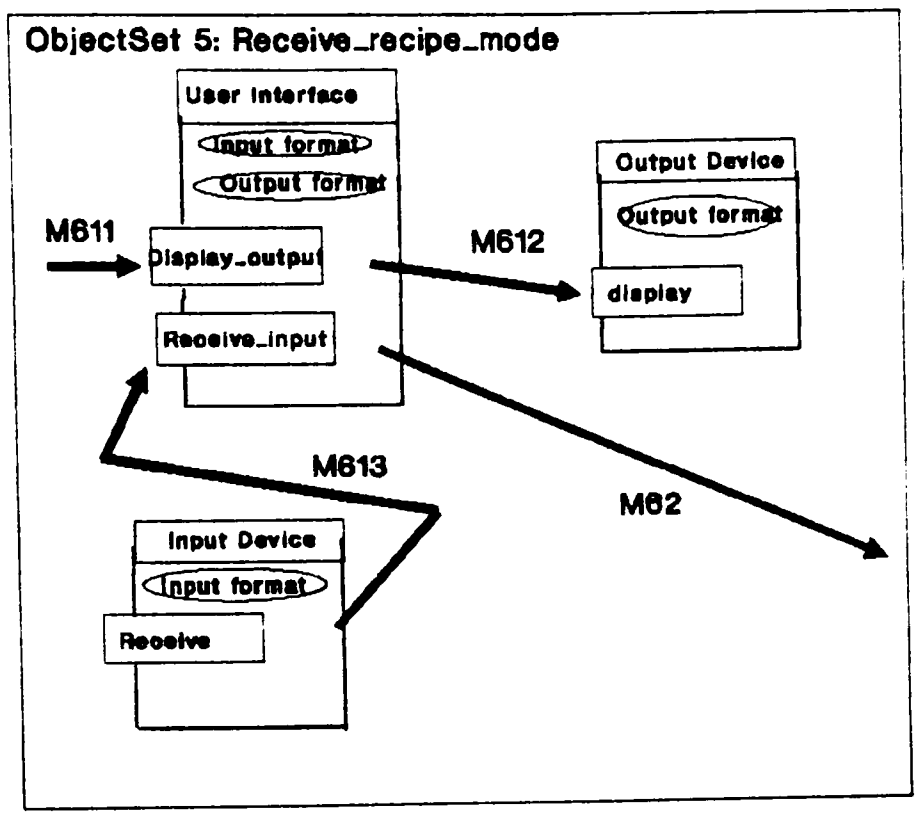


Figure 4.6. Continued.

(e)



(f)

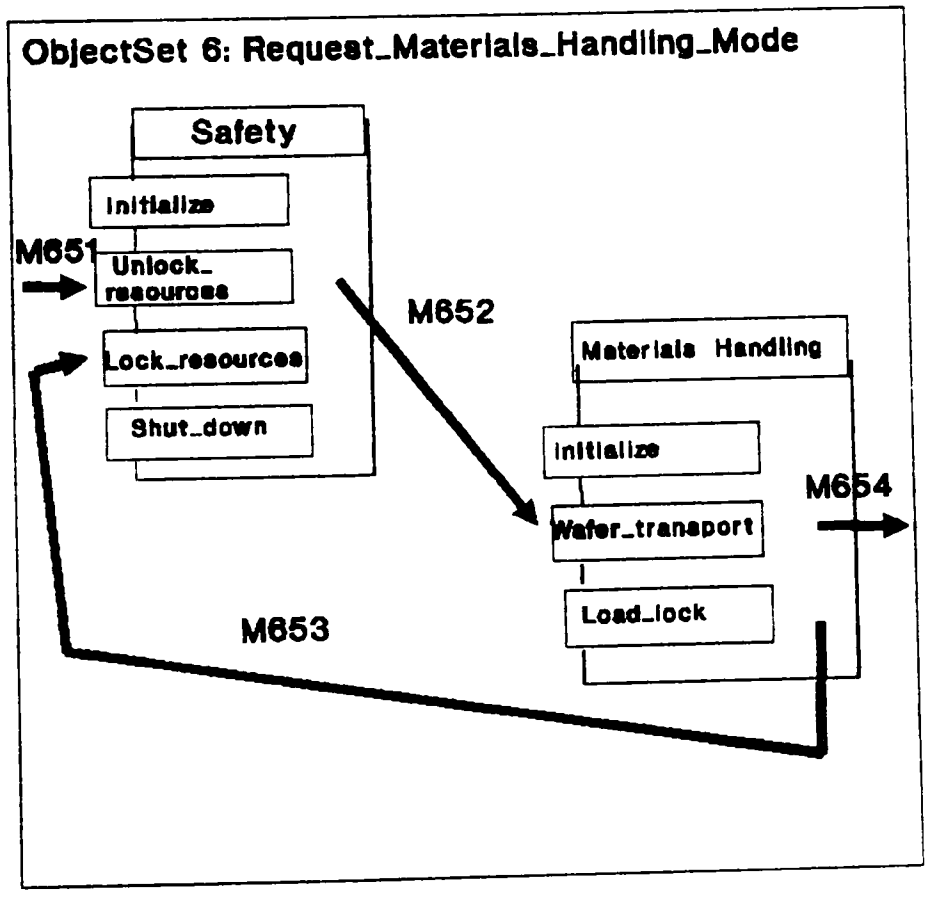
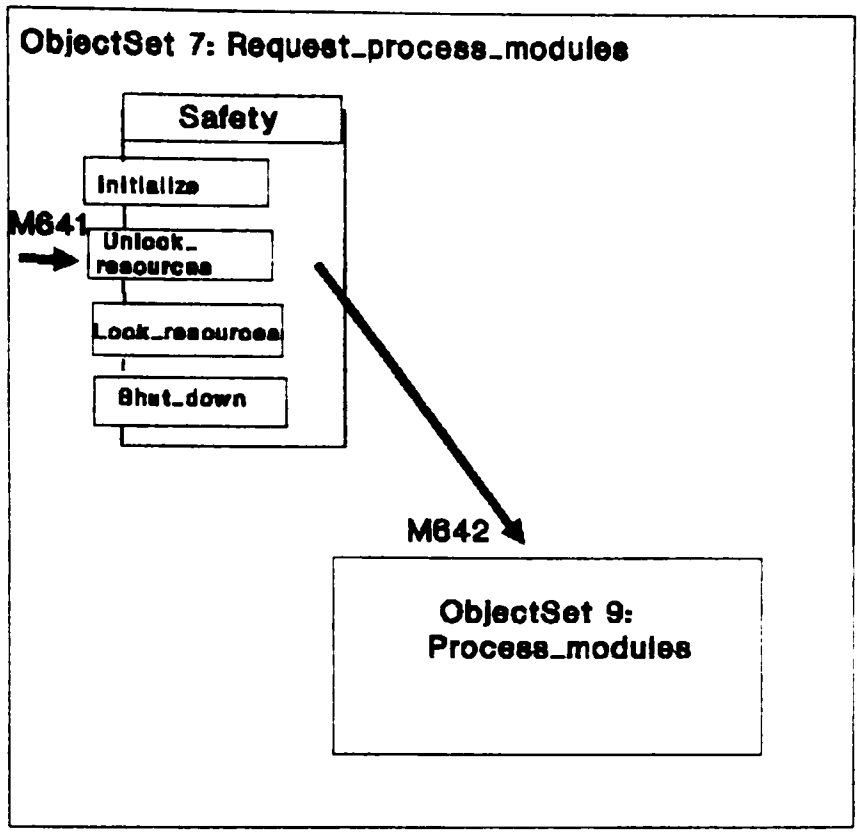


Figure 4.6. Continued.

(g)



(h)

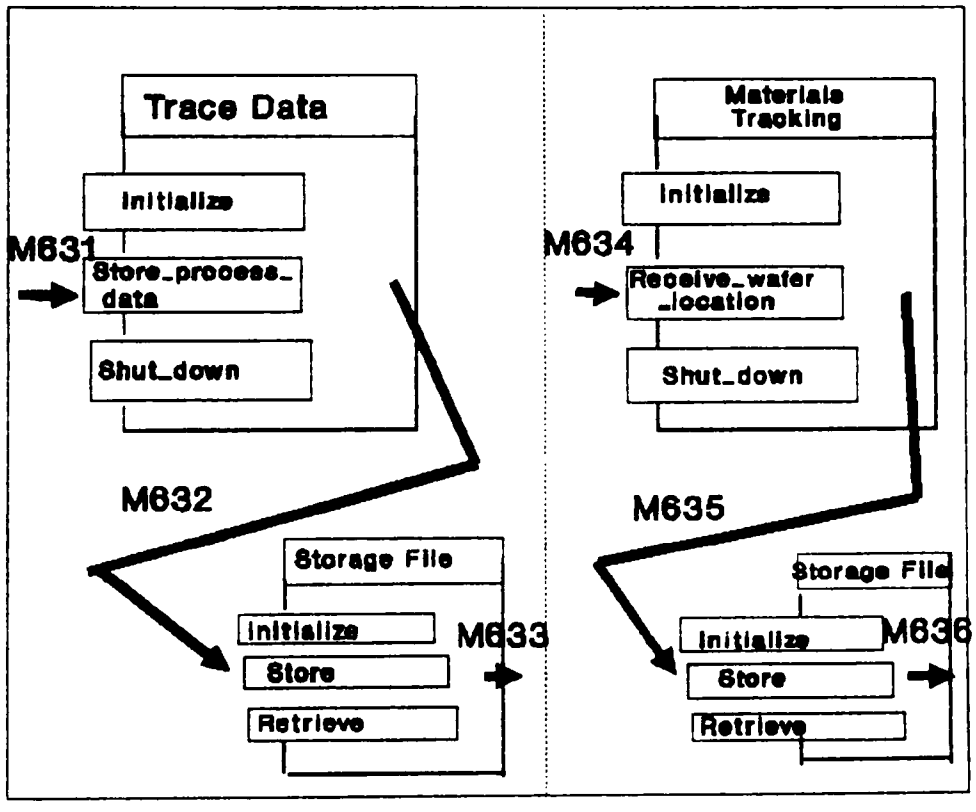
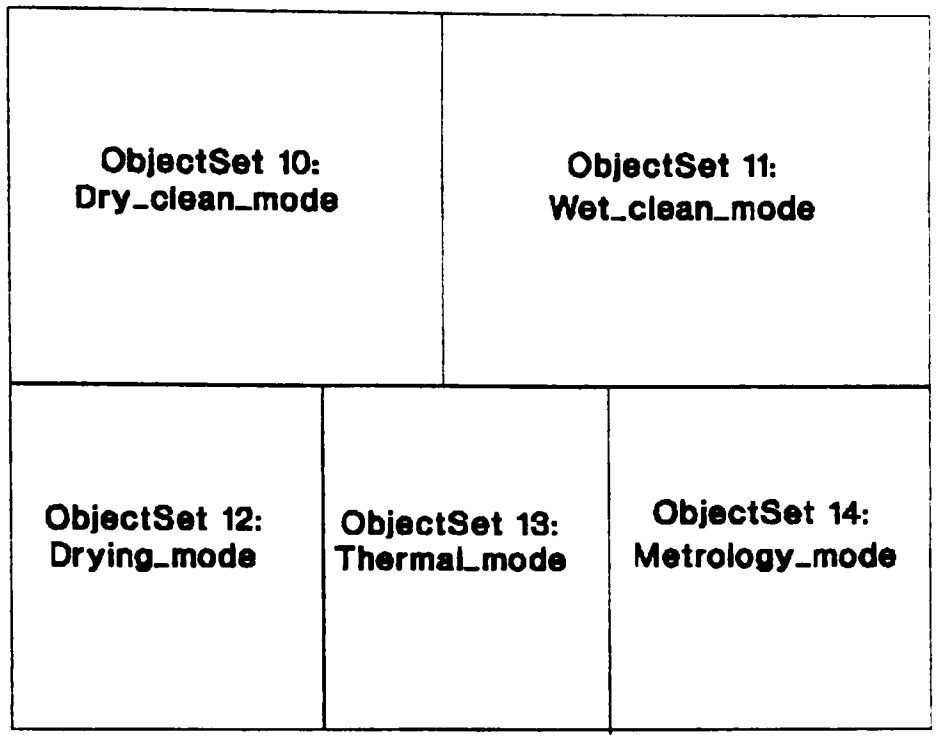


Figure 4.6. Continued.

(i)



(j)

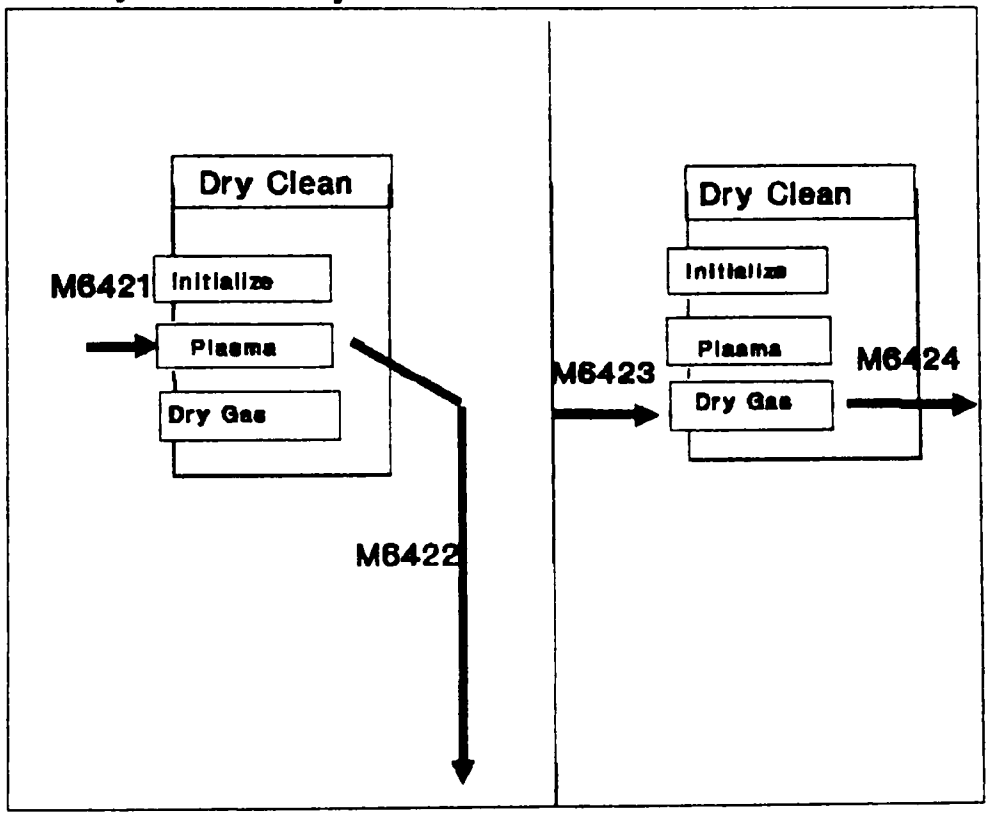
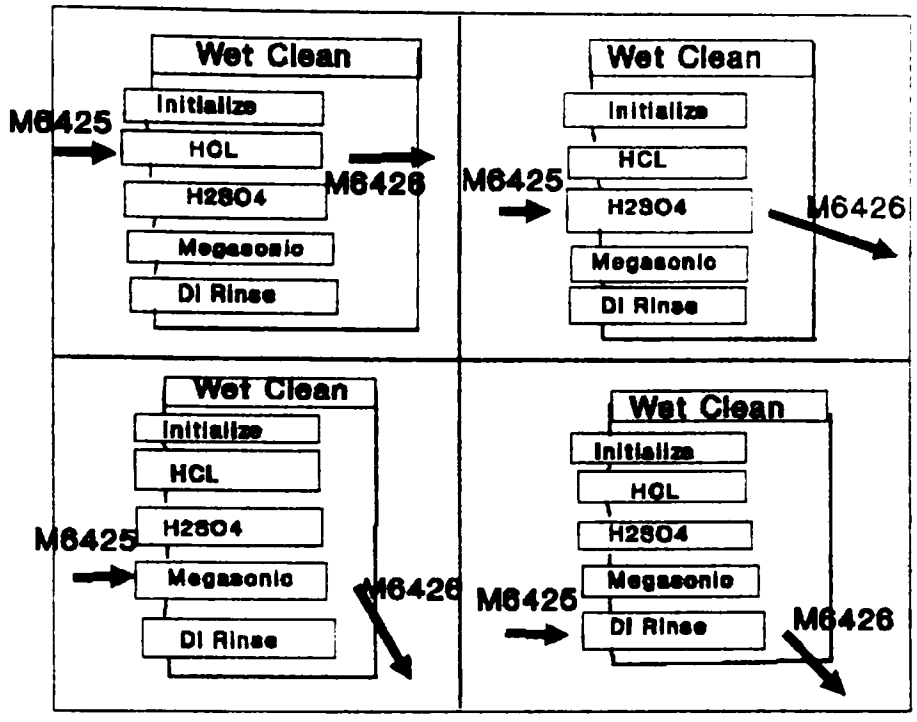


Figure 4.6. Continued.

(k)



(l)

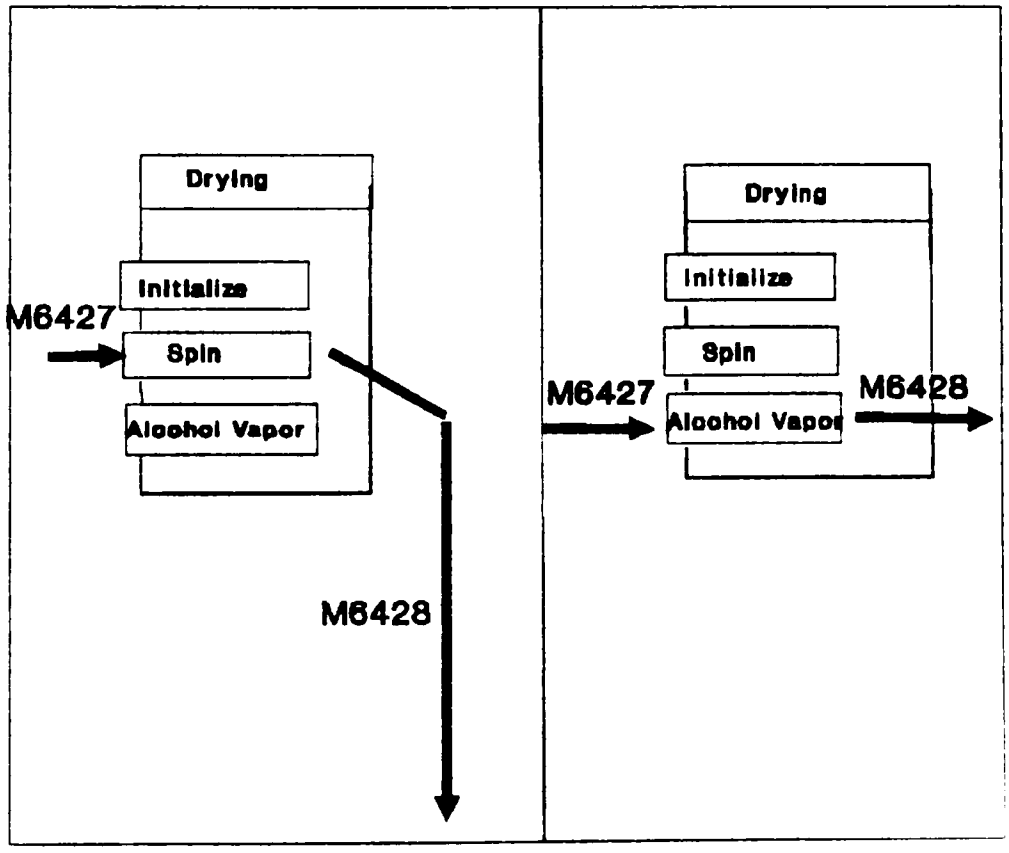
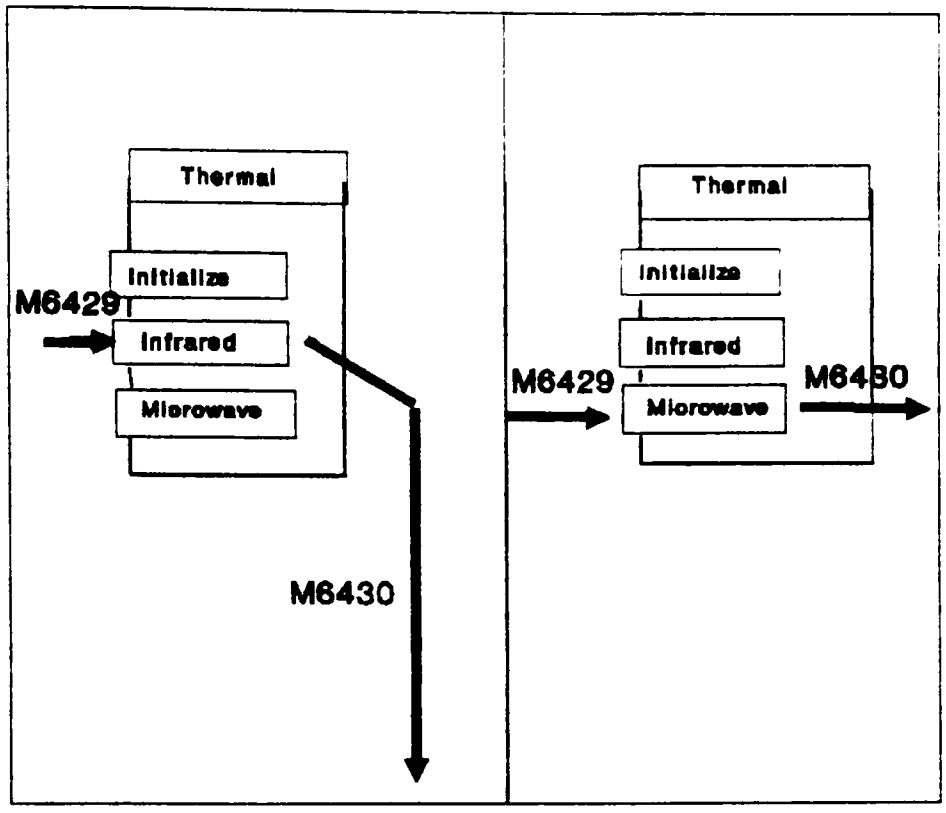


Figure 4.6. Continued.

(m)



(n)

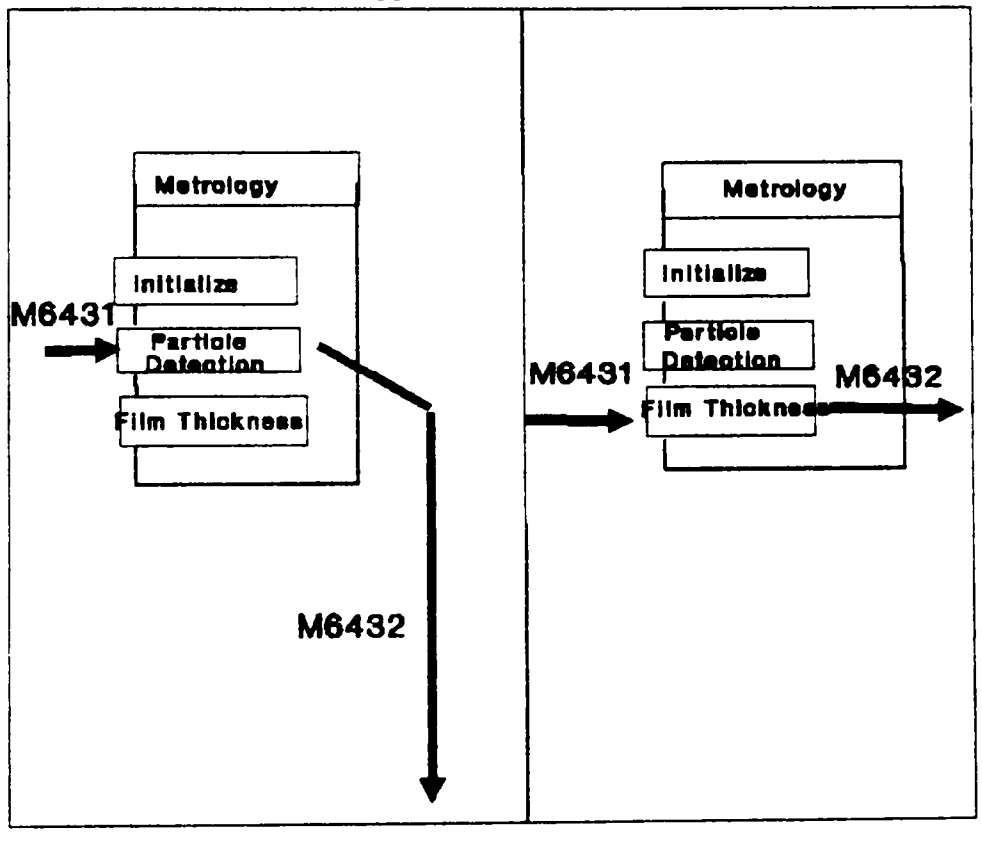
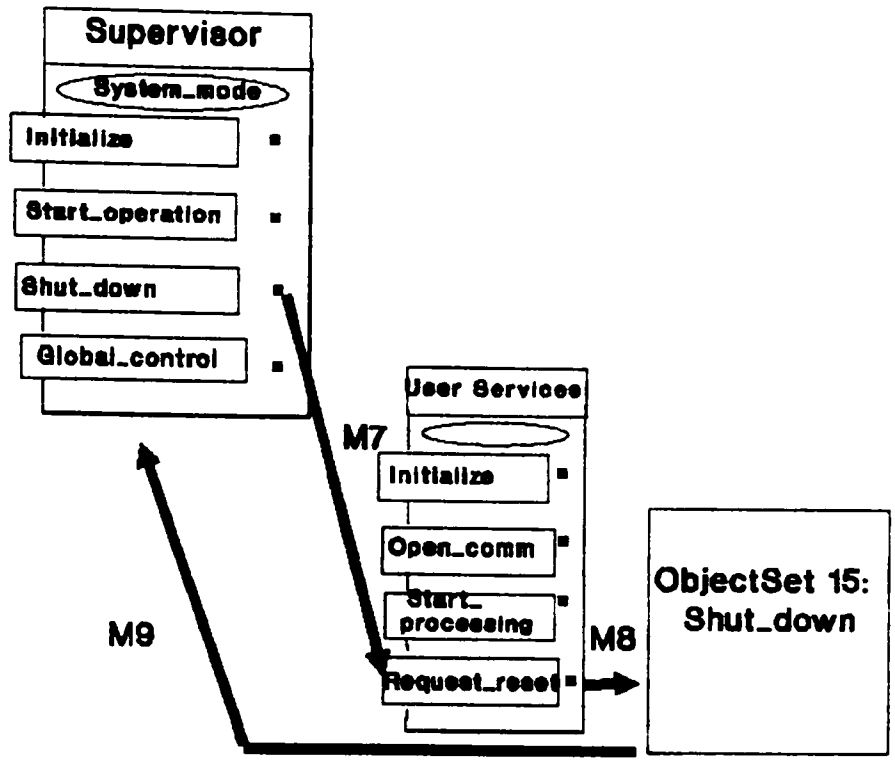


Figure 4.6. Continued.

(a)



ObjectSet 15: Shut_down

(b)

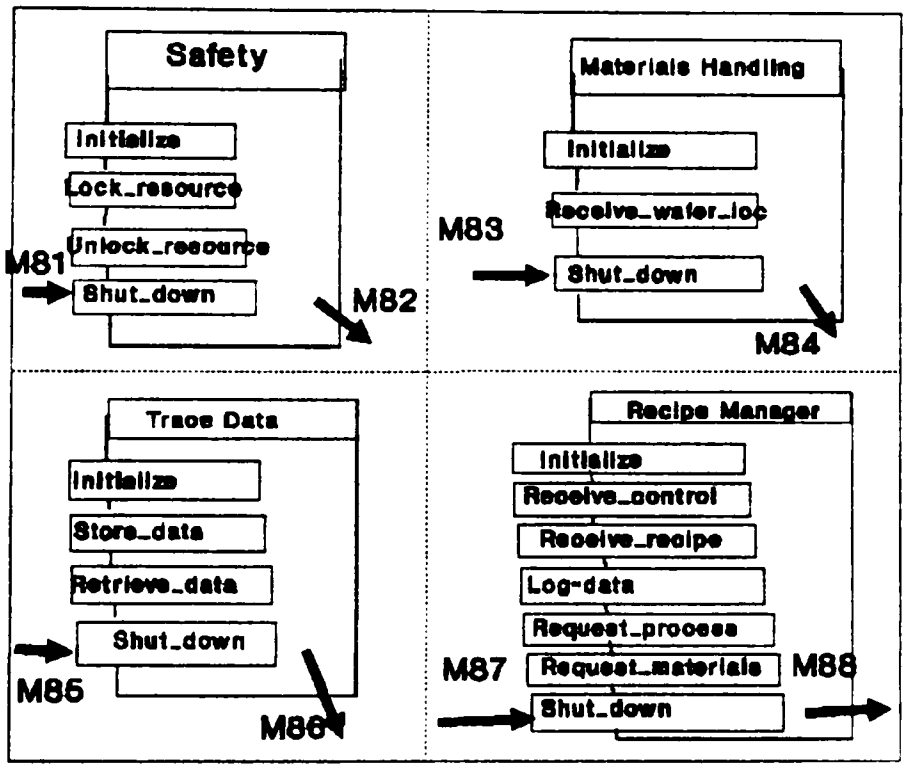


Figure 4.7. Message passing diagrams for shutdown of the single wafer cleaning system.

A message dictionary in Figures 4.8 and 4.9 keeps track of all messages which are shown in the message passing diagrams for the single wafer cleaning system: Figure 4.5, Figure 4.6, and Figure 4.7. Figure 4.8 shows the following information for each messages: message number, the identifiers of the source and destination objects, requested method, and parameters if needed. Due to the limited space, the priority value and time constraints for each message are put in Figure 4.9.

Message Number	Source Object	Destination Object	Message Contents	Parameters
M1	Supervisor	ObjectSet1: All_Objects	Initialize	
M11	Supervisor	User_Interface (ObjectSet1)	Initialize	
M12	Supervisor	Log (ObjectSet1)	Initialize	
M13	Supervisor	Process_Modules (ObjectSet1)	Initialize	
.				
.				
.				
M2	ObjectSet1	Supervisor	(Notification) Result	
M3	Supervisor	User_Services	Open_communi _cation	

Figure 4.8. Message dictionary (Part 1) of the single wafer cleaning system.

Message Number	Source Object	Destination Object	Message Contents	Parameters
M4	User_Services	User_Interface (ObjectSet2: Start_Mode)	Display_output	Login _prompt
M41	User _Interface (ObjectSet2)	Output_Device (ObjectSet2)	Display	Login _prompt
M42	Input_Device (ObjectSet2)	User_Interface (ObjectSet2)	Receive _input	Operator _Id/pass _word
M43	User _Interface (ObjectSet2)	Security (ObjectSet2)	Verify	Id/password
M5	User _Services	ObjectSet3: (Load_Wafer _Mode)		
M51	User _Services	User_Interface (ObjectSet3)	Display _output	Prompt _to_place _wafer
M52	User _Services	Material _Handling (ObjectSet3)	Wafer _transport	
M53	User _Interface (ObjectSet3)	Output_Device (ObjectSet3)	Display	Prompt _to_place _wafer
M54	Materials _Handling (ObjectSet3)	User_Services	(Notification)	Result
M6	User _Services	Recipe_Manager (ObjectSet4: Process_Mode)	Receive _control	

Figure 4.8. Continued.

Message Number	Source Object	Destination Object	Message Contents	Parameters
M61	Recipe _Manager (ObjectSet4)	ObjectSet5: Receive_Recipe _Mode		
M611	Recipe _Manager (ObjectSet4)	User_Interface (ObjectSet5)	Display _output	Recipe _control _menu
M612	User _Interface (ObjectSet5)	Output_Device (ObjectSet5)	Display	Recipe _control _menu
M613	Input _Device (ObjectSet5)	User_Interface (ObjectSet5)	Receive _input	Recipe
M62	User _Interface (ObjectSet5)	Recipe_Manager (ObjectSet4)	(Notification)	Recipe
M63	Recipe _Manager (ObjectSet4)	ObjectSet8: Log_Data		
M631	Recipe _Manager (ObjectSet4)	Trace_Data (ObjectSet8)	Store _process _data	Process _data
M632	Trace_Data (ObjectSet8)	Storage_File (ObjectSet8)	Store _data	Process _data
M633	Storage _File (ObjectSet8)	Recipe_Manager (ObjectSet4)	(Notification)	Result
M634	Recipe _Manger (ObjectSet4)	Materials _Tracking (ObjectSet8)	Receive _wafer _location	Wafer _loc Wafer _num

Figure 4.8. Continued.

Message Number	Source Object	Destination Object	Message Contents	Parameters
M635	Materials _Tracking (ObjectSet8)	Storage_File (ObjectSet8)	Store _data	Wafer _loc Wafer _num
M636	Storage _File (ObjectSet8)	Recipe_Manager (ObjectSet4)	(Notification)	Result
M64	Recipe _Manager (ObjectSet4)	ObjectSet7: Request_Process _Modules		
M641	Recipe _Manager (ObjectSet4)	Safety (ObjectSet7)	Unlock _resources	Resource
M642	Safety (ObjectSet7)	ObjectSet 9: Process_Modules		
M6421	Safety (ObjectSet7)	Dry_Clean (ObjectSet10: Dry_Clean_Mode)	Plasma	Selection
M6422	Dry_Clean (ObjectSet10)	Plasma (ObjectSet7)	...	Selection
M6423	Safety (ObjectSet7)	Dry_Clean (ObjectSet10)	Dry_Gas	Selection
M6424	Dry_Clean (ObjectSet10)	Dry_Gas (ObjectSet7)	...	Selection
M6425	Safety (ObjectSet7)	Wet_Clean (ObjectSet11: Wet_Clean_Mode)	...	Selection
M6426	Wet_Clean (ObjectSet11)	Safety (ObjectSet7)	(Notification)	Result

Figure 4.8. Continued.

Message Number	Source Object	Destination Object	Message Contents	Parameters
M6427	Safety (ObjectSet7)	Drying_Mode (ObjectSet12: Drying_Mode)	...	Selection
M6428	Drying_Mode (ObjectSet12)	Safety (ObjectSet7)	(Notification)	Result
M6429	Safety (ObjectSet7)	Thermal (ObjectSet 13: Thermal_Mode)	...	Selection
M6430	Thermal (ObjectSet13)	Safety (ObjectSet7)	(Notification)	Result
M6431	Safety (ObjectSet7)	Metrology (ObjectSet14: Metrology_Mode)	...	Selection
M6432	Metrology (ObjectSet14)	Safety (ObjectSet7)	(Notification)	Result
M65	Recipe _Manager (ObjectSet4)	ObjectSet 6: Request_Materials _Handling_Mode		
M651	Recipe _Manager (ObjectSet4)	Safety (ObjectSet6)	Unlock _resource	resource
M652	Safety (ObjectSet6)	Materials _Handling (ObjectSet6)	Wafer _transport	Wafer_num Wafer_loc
M653	Materials _Handling (ObjectSet6)	Safety (ObjectSet6)	Lock _resource	resource
M654	Materials _Handling (ObjectSet6)	Recipe _Manager (ObjectSet4)	(Notification)	Result

Figure 4.8. Continued.

Message Number	Source Object	Destination Object	Message Contents	Parameters
M7	Supervisor	User_Services	Request_reset	
M8	User_Services	ObjectSet15: Shut_Down		
M81	User_Services	Safety (ObjectSet15)	Shut_down	
M82	Safety (ObjectSet15)	Supervisor	(Notification) Result	
M83	User_Services	Materials _Handling (ObjectSet15)	Shut_down	
M84	Materials _Handling (ObjectSet15)	Supervisor	(Notification) Result	
M85	User_Services	Trace_Data (ObjectSet15)	Shut_down	
M86	Trace_Data (ObjectSet15)	Supervisor	(Notification) Result	
M87	User_Services	Recipe _Manager (ObjectSet15)	Shut_down	
M88	Recipe _Manager (ObjectSet15)	Supervisor	(Notification) Result	
M9	ObjectSet15	Supervisor	(Notification) Result	

Figure 4.8. Continued.

Message Number	Priority Value	Time Constraints
M1	110	SC: ◇ System_mode = initialize BT: 2.0s ET: 5.0s
M2	50	SC: ◇ Status = initialized BT: 3.0s ET: 5.0s
M3	100	SC: ◇ Received M2 from all objects system_mode = operation BT: 1.0s ET: 2.0s
M41	95	SC: Output_device_status = ready BT: 1.0s ET: 2.0s
M42	90	SC: ◇ Received input from operator BT: .. ET: ..
Mx	100	SC: Operator_Id/password = verified BT: 3.0s ET: 5.0s
M6	100	SC: Wafers are loaded & available BT: .. ET: ..
M7	120	SC: System_mode = shutdown BT: 2.0s ET: 5.0s
M83	100	SC: Process_status = Completed: DI_water_rinse and Dry BT: 5.0s ET: 10.0s

Figure 4.9. Message dictionary (Part 2) of the single wafer cleaning system.

As mentioned earlier, the priority value is denoted as any integer, with the higher the integer, the greater the priority. In this case study, a range of values between 60 and 120 is used to indicate the normal operation invocation messages. The notification messages are given the values between 20 and 59. Even though any error messages and the global control messages from the supervisor object are not shown explicitly, they are assumed to have the higher range of priority values (that is, higher than 120). The following codes are used to indicate the appropriate elements of the time constraints: SC for Starting Conditions, BT for Begin Time, and ST for Stop Time. The begin and stop time can be specified by a real number followed by any specific time units: "h" for hour, "m" for minute, "s" for second, "ms" for millisecond, and "us" for microsecond. For example, the notation of "BT: 5.0ms" specifies that the execution should start in 5 milliseconds. Figure 4.9 shows the priority values and time constraints for some of the messages in the single wafer cleaning system. Even though the time constraints for each requested operations are evaluated and analyzed in the analysis phase, it is in the design and implementation phases that they are finally solidified.

Message Passing Scenario for the Given Recipe

A recipe contains the predefined steps for cleaning wafers. Among the various process methods such as dry clean, wet clean, drying, thermal, and metrology, a set of methods is pre-stored as a specific recipe with a unique recipe number (or name), so that the operator does not have to organize the combination of wafer cleaning methods, but only select any existing recipe which has the desired cleaning steps in it. In this scenario, a recipe which has the following steps will be selected by the operator:

1. Dry Clean using Plasma (Microwave)
2. Wet Clean with HCL
3. Drying using Alcohol Vapor
4. Metrology for particle detection
5. Dry Clean using Dry Gas (Ozone) if the high level of particles are detected in Step 4.

After the system is initialized and the operator is granted access, a cassette of wafers is loaded and ready for processing. Then, the Recipe Manager receives control and gets the above recipe selected by the operator. For this recipe, the following messages must be passed for each wafer.

1. The Recipe Manager sends a message to the Materials Handling to load the next wafer in the Dry Clean process module. This message goes through the Safety to make sure that the designated process module is empty. These messages are passed in ObjectSet6 (Request_Materials_Handling_Mode).

2. The Recipe Manager sends a message to the Dry Clean to activate the method 'Plasma' which in turn sends a message to the Plasma with a selection of Microwave. These messages are passed in ObjectSet7 and ObjectSet10 which are Request_Process_Modules and Dry_Clean_Mode, respectively.

3. The Recipe Manager sends a message to Trace Data and Materials Tracking, and they in turn send messages to the Storage file to store wafer processing data. These message passing activities are done in ObjectSet8 (Log_Data).

4. After receiving the notification message that the dry clean is completed, the Recipe Manager sends a message to the Materials Handling to unload the wafer from the Dry Clean module and load in the Wet clean module for the next cleaning step. This message goes through the Safety to unlock and lock the resources appropriately (Done in ObjectSet6).

5. The Recipe Manager sends a message to the Wet Clean to activate the method 'HCL.' This message passing is done in ObjectSet11 (Wet_Clean_Mode).

6. The Recipe Manager sends a message to Trace Data and Materials Tracking, and they in turn send messages to the Storage file to store wafer processing data. These message passing activities are done in ObjectSet8 (Log_Data).

7. After receiving the notification message that the wet clean is done, the Recipe Manager sends a message to the Materials Handling to unload the wafer from the Wet Clean module and load in the Drying module through the Safety (Done in ObjectSet6).

8. The Recipe Manager sends a message to the Drying to activate the method 'Alcohol Vapor.' This is done in ObjectSet12 (Drying_Mode).

9. The Recipe Manager sends a message to Trace Data and Materials Tracking, and they in turn send messages to the Storage file to store wafer processing data. These message passing activities are done in ObjectSet8 (Log_Data).

10. After receiving the notification message that the drying is completed, the Recipe Manager sends a message to the Materials Handling to unload the wafer

from the Drying module and load in the Metrology through the Safety (Done in ObjectSet6).

11. The Recipe Manager sends a message to the Metrology to activate the method 'Particle Detection.' This message passing is done in ObjectSet14 (Metrology_Mode).

12. The Recipe Manager sends a message to Trace Data and Materials Tracking, and they in turn send messages to the Storage file to store wafer processing data. These message passing activities are done in ObjectSet8 (Log_Data).

13. After receiving the notification message that the metrology is done, the Recipe Manager sends a message to the Materials Handling to unload the wafer from the Metrology, and load it in the Dry Clean module if the result of metrology shows the higher level of particles detected. Otherwise, the recipe is completed and the wafer has finished being processed (Done in ObjectSet6).

14. If the wafer is loaded in the Dry Clean module, the Recipe Manager sends a message to the Dry Clean to activate the method 'Dry Gas' which in turn sends a message to the Dry Gas to active the method 'Ozone.' This message passing is done in ObjectSet10 (Dry_Clean_Mode).

15. After receiving the notification message that the dry clean is completed, the Recipe Manager sends a message to Material Handling to unload the wafer (Done in ObjectSet6).

16. The Recipe Manager sends a message to Trace Data and Materials Tracking, and they in turn send messages to the Storage file to store wafer processing data. These message passing activities are done in ObjectSet8 (Log_Data).

All processing is done on a single wafer basis until all the wafers in a cassette will be processed.

CHAPTER V

EVALUATION

5.1 Evaluating Criteria

To judge the resulting specification, the following criteria for evaluating a technique developed by Davis [10] and the expected results for each criteria (shown inside the parentheses) were presented in the proposal of this research:

1. When the technique is properly used, the resulting specification should be helpful and understandable to non-computer-oriented customer and users. (above 9)

2. When the technique is properly used, the resulting specification should be able to serve effectively as the basis for design and testing. (between 7-9)

3. The technique should provide automated checks for ambiguity, incompleteness, and inconsistency. (N/A)

4. The technique should encourage the requirements writer to think and write in terms of external product behavior, not internal product components. (above 7)

5. The technique should help organize the information in the specification. (above 9)

6. The technique should provide a basis for automated prototype generation. (N/A)

7. The technique should provide a basis for automated system test generation. (N/A)

8. The technique should be suitable to the particular application.

This scale is from 0 to 10, with 0 being poor and 10 being excellent. Automated checking, prototype generation, and automatic test generation are beyond the scope of this research since only the concepts of the OOART are introduced and this technique has not yet been automated. However, such automation of the OOART is possible, and is discussed in Chapter 6. A (subjective) evaluation of the OOART, using Davis' criteria, is below.

5.2 Evaluation Results

Understandable to Computer-Naive Personnel

As Coad and Yourdon claim, the object-oriented approach supports the basic principles that people use to organize the problem space: the differentiation into particular objects and their attributes, the distinction between whole objects and their component parts, and the distinction between different classes of objects [26]. In the object-oriented approach, real-world entities are

directly mapped onto objects in the model world [62]. Therefore, the process of decomposing the system into objects can be easily understood by non-computer-oriented personnel.

A graphical representation of the system can be more easily understood than a textual representation. In the OOART, a set of graphical notations are used to represent the requirements specification of the system: abstraction layer diagram, inheritance diagram, message passing diagram, object dictionary and message dictionary. As Davis states, understandability appears to be inversely proportional to the level of complexity and formality present [10]. Therefore, the OOART is using the different layers of abstraction to reduce the amount of complexity that must be comprehended at one time and to increase the understandability of the specification. Evaluation Result: 9.

Basis for Design and Test

The requirements specification should not only be understandable to computer-naive personnel (a customer who initially signs off the system), but must also be unambiguous and understandable to the computer scientists who will design, implement, and test the system based on

this specification. The formal model developed using the OOART allows the analyst to unambiguously specify every objects in the system and any potential message passing activities between them. Therefore, the specification can define the intended product behavior, and thus serve as a basis for both design and test.

In the object-oriented approach, a system is viewed in terms of objects only. Objects are the items of interest in all phases of the software life cycle. Also, the same paradigms can be applied to all phases of the life cycle, resulting the seamless transformation process between the phases. As Constantine claims, the models developed in the object-oriented analysis and design can be expressed in the same or equivalent notations based on common principles [62]. Evaluation Result: 9.

External View, Not Internal View

Using the object-oriented approach, the external entities are first defined, and then the attributes and operations for each entities are described. The concept of information hiding allows the encapsulation of the internal workings of each objects and provides the external view on each object's behavior. Even though the internal structures of an object and the message queue

are discussed in detail previously, this explanation has been included to display the underlying concepts of the OOART. Therefore, the resulting specification of the OOART does not show any internal structures of objects. Instead, each object reveals only its unique identifier and the available methods to other objects. Every possible external behaviors of objects in the system are represented in the message passing diagrams of the OOART. Evaluation Result: 8.

Organizational Assistance

The OOART helps organize the information in the specification by providing the different layers of abstractions in the abstraction layer diagram, and the objectsets in the message passing diagram. The process of building the layers of abstraction organizes the information hierarchically. An objectset groups the objects which are involved in the similar tasks together to organize the behavioral requirements. Also, the object and message dictionaries provide a systematic way of keeping track of the information on every objects and messages in the system. Evaluation Result: 8.

Appropriate Applications

The OOART has been developed for real-time systems where concurrency and the time constraints are the important characteristics of the system. To support the concurrency of the system, a set of objects is grouped together to form an "objectset"; then, any objectsets which would be processed concurrently are represented as separate boxes from each other by dashed lines in the message passing diagram. Therefore, communication between concurrent objects can be denoted without ambiguity.

Time constraints are incorporated into the OOART by embedding them in messages. When an object sends a message to other objects, the message contains the time constrains for the specified operation. This information is compared against a worst case execution time which is attached to each operation to decide whether the requested operation can be finished within the specified time. The OOART also supports complex real-time systems by providing a set of notations that facilitates the hierarchical decomposition of the system (abstraction layer) and the grouping of related objects together (objectsets).

CHAPTER VI

CONCLUSIONS

In this research, an object-oriented analysis methodology which can be applied to real-time systems has been developed and explored. This methodology, as in other object-oriented approaches, is based on the concept of an object. To explain the basic workings of objects in the OOART, the internal structure of an object has been explored in detail, along with message passing techniques which employ a priority queue in each object to handle message traffic. To support time constraints, which is an important characteristic of real-time systems, the OOART embeds this information in each message.

To represent the OOART in the resulting requirements specification, a set of graphical notations which not only supports the concept of the object-oriented approach, but which also supports the characteristics of real-time systems, has been utilized. As a case study of this research, the OOART is applied to the single wafer cleaning system which implements a distributed real-time control of semiconductor process equipment. In this case study, a supervisor object is used, instead of having a

cluster controller which manages inter-node communication of the objects in the system. This supervisor object performs the initialization and shutdown of the system, and gives a semi-global coordination to the objects.

In object-oriented software development, objects are the items of interest in all phases of the software life cycle. The same paradigms can be applied to all phases of the life cycle since the underlying concept is same. Therefore, the transition flows smoothly in a software life cycle. The notations and conventions employed in the OOART might also be used, or served as a basis for the object-oriented design.

Using the evaluation criteria developed by Davis [10], the resulting specification of the OOART has been evaluated. As a result, the OOART claims to generate the requirements specification which is understandable to computer-naive personnel, serves as a basis for design and test, provides an external view, and assists to organize the information in the system. However, only the concepts are introduced in this research; therefore, any evaluation regarding any automated support has been omitted.

Even though many current CASE products are based on the structured analysis, some CASE tools for the object-

oriented analysis have already been developed and several CASE vendors are planning to develop object-oriented analysis CASE tools [64]. If the OOART could be supported by CASE tools, the maintenance of the object and message dictionaries would be easier and more consistent, and the automatic checking of errors such as duplicate object identifiers or message numbers would be detected easily. The capabilities of the automatic prototypes and test generation might also be incorporated into the CASE tools.

REFERENCES

[1] B. Stroustrup, "What is Object-Oriented Programming?," IEEE Software, vol. 5, no. 3, pp. 10-20, May 1988.

[2] R. Sebesta, Concepts of Programming Languages. 1st edition, 1989, The Benjamin/Cummings Publishing Co.

[3] E. Seidewitz and M. Stark, "Towards a general object-oriented software development methodology," Ada Letters, pp. 54-67, July/August 1987.

[4] G. Booch, "Object-Oriented Development," IEEE Transactions on Software Engineering, vol. 12, no. 2, pp. 211-221, February 1986.

[5] S. Boyd, "Object-Oriented Design and PAMELA: A comparison of two design methods for Ada," Ada Letters, pp. 68-78, July/August 1987.

[6] P. Ward, "How to Integrate Object Orientation with Structured Analysis and Design," IEEE Software, vol. 6, no. 2, pp. 74-82, March 1989.

[7] S. Bailin, "An Object-Oriented Requirements Specification Method," Communications of the ACM, vol. 32, no. 5, pp. 608-623, May 1989.

[8] S. Yau and J. Tsai, "A survey of Software Design Techniques," IEEE Transactions on Software Engineering, vol. 12, no. 6, pp. 713-721, June 1986.

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

[10] A. Davis, "A Comparison of Techniques for the Specification of External System Behavior," Communications of the ACM, vol. 31, no. 9, pp.1098-1115, September 1988.

[11] D. Halbert and P. O'Brien, "Using Types and Inheritance in Object-Oriented Programming," IEEE Software, pp. 71-79, September 1987.

[12] S. Hufnagel and J. Browne, "Performance Properties of Vertically Partitioned Object-Oriented Systems," IEEE Transactions on Software Engineering, vol. 15, no. 8, pp. 935-946, August 1989.

[13] W. Harrison, J. Shilling, and P. Sweeney, "Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm," OOPSLA '89 Proceedings, pp. 85-94, October 1-6, 1989.

[14] W. Rosenblatt, J. Wileden , and A. Wolf, "OROS: Toward a Type Model for Software Development Environments," OOPSLA '89 Proceedings, pp. 297-304, October 1-6, 1989.

[15] D. Parnas, "On the Criteria to be used in Decomposing Systems into Modules," Communications of the ACM, December 1972.

[16] J. Leite, A Survey on Requirements Analysis. RTP 071, University of California, Irvine, June 1987.

[17] W. Wolf, "A Practical Comparison of Two Object-Oriented Languages," IEEE Software, vol. 6, no. 5, pp.61-68, September 1989.

[18] M. Shaw, "Abstraction Techniques in Modern Programming Languages," IEEE Software, vol. 1, no. 4, p. 10, October 1984.

[19] P. Yelland, "First Steps Towards Fully Abstract Semantics for Object-Oriented Languages," The Computer Journal, vol. 32, no. 4, pp.290-296, 1989.

[20] K. Lieberherr and I. Holland, "Assuring Good Style for Object-Oriented Programs," IEEE Software, vol. 6, no. 5, pp. 38-48, September 1989.

[21] Smalltalk/V 286 Tutorial and Programming Handbook. May 1988, Digitalk Inc.

[22] O. Dahl, "Object-Oriented Specification," in Research Directions in Object-Oriented Programming. B.Shriver and P.Wegner, ED. Cambridge, MA.: The MIT Press,1987, pp.561-576.

[23] S. Hekmatpour and D. Ince, Software Prototyping, Formal Methods and VDM, Wokingham,England: Addison-Wesley Publishing Company, 1988.

[24] N. Gehani and A. McGettrick, ED. Software Specification Techniques, Wokingham, England: Addison-Wesley Publishing Company, 1985.

[25] R. Abbott, An Integrated Approach to Software Development, New York, NY: John Wiley & Sons, 1986.

[26] P. Coad and E. Yourdon, Object-Oriented Analysis, Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1990.

[27] J. Brackett, Software Requirements, SEI curriculum module SEI-CM-19-1.0, SEI, Carnegie Mellon University, December 1988.

[28] S. Shlaer and S. Mellor, Object-Oriented Systems Analysis: Modeling the World in Data, Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1988.

[29] S. Woodfield, D. Embley and B. Kurtz, "Extending Analysis Paradigms," IEEE Conference Proceedings, pp.441-446.

[30] W. Marcy, "Object-Oriented Analysis for Semiconductor Process Control," Proc. SME Semiconductor Material Handling and Process Control Technology for Advanced Manufacturing, April 1990, San Jose, California.

[31] W. Marcy and D. Bagert, "A Distributed Real Time Control and Operating System Design for Single Wafer Cleaning Systems: Requirements Analysis," Texas Tech University, technical report, August 21, 1989.

[32] T. DeMarco, Structured Analysis and System Specification, Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1979.

[33] C. Gane and T. Sarson, Structured Systems Analysis: Tools and Techniques, Englewood Cliffs, NJ; Prentice-Hall, 1979.

[34] S. McMenamins and J. Palmer, Essential Systems Analysis, New York: Yourdon Press, 1984.

[35] D. Robson, "Object-Oriented Software Systems," BYTE, pp.5-8, August 1981.

[36] R. Pressman, Software Engineering: A Practitioner's Approach, New York: McGraw-Hill, 1987.

[37] D. Hatley and I. Pirbhai, Strategies for Real-Time System Specification, New York, NY; Dorset House Publishing, 1987.

[38] S. Allworth and R. Zobel, Introduction to Real-Time Software Design. 2nd edition, Springer-Verlag New York Inc., 1987.

[39] A. Burns and A. Wellings, Real-Time Systems and Their Programming Languages, Wokingham, England; Addison-Wesley Publishing Company, 1990.

[40] M. Blackman, The Design of Real-Time Applications, London, England; John Wiley & Sons, 1975.

[41] W. Marcy, "Using Finite State Machines to Develop Specifications For Process Modules and Process Program," Texas Tech University, technical report, February 19, 1990.

[42] C. Davis and C. Vick, "The Software Development System," IEEE Transactions on Software Engineering, vol. 3, no. 1, pp.69-84, January 1977.

[43] M. Alford, "SREM at the Age of Eight; The Distributed Computing Design System," IEEE Computer, vol. 18, no. 4, pp. 36-46, April 1985.

[44] M. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, vol. 3, no. 1, pp.60-69, January 1977.

[45] A. Davis, "The Design of a Family of Application-Oriented Requirements Languages," IEEE Computer, vol. 15, no. 5, pp. 21-28, May 1982.

[46] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Transactions on Software Engineering, vol. 8, no. 3, pp.250-269, May 1982.

[47] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," IEEE Transactions on Software Engineering, vol. 12, no. 2, pp.312-325, February 1986.

[48] J. Northcutt, Mechanisms for Reliable Distributed Real-Time Operating Systems -- The Alpha Kernel: Perspectives In Computing, vol.16, W. Rheinboldt and D. Siewiorek, ED, Boston, MA: Academic Press, Inc., 1987.

[49] S. Levi and A. Agrawala, Real Time System Design, New York, NY: McGraw-Hill Publishing Company, 1990.

[50] P. Jorgensen, Real-Time System Requirements Specification: Issues for CASE, International Phoenix Conference on Computers and Communications, Scottsdale, AZ, March 21-24, 1990.

[51] P. Helman and R. Veroff, Intermediate Problem Solving and Data Structures, Menlo Park, CA: The Benjamin/ Publishing Company, Inc., 1986, pp.463-475.

[52] D. Christodoulakis, Computing Reviews, (Review of [7]) p. 203, April 1990.

[53] T. Korson and J. McGregor, "Understanding Object-Oriented: A Unifying Paradiam," Communications of the ACM, vol. 33, no. 9, pp. 40-60, September 1990.

[54] B. Henderson-Sellers and J. Edwards, "The Object-Oriented Systems Life Cycle," Communications of the ACM, vol. 33, no. 9, pp.142-159, September 1990.

[55] L. Deimel, A Brief Introduction to Temporal Logic for Software Engineer, SEI Carnegie Mellon University, March 1990.

[56] W. Wood, Temporal Logic Case Study, SEI Carnegie Mellon University: Technical Report CMU/SEI-89-TR-24, August 1989.

[57] D. Durant, G. Carlson, and P. Yao, Programmer's Guide to Windows, San Francisco, CA: Sybex, 1987.

[58] S. Levi, A Methodology for Designing Distributed, Fault-Tolerant, and Reactive Real-Time Operating System, Ph.D. Dissertation, Department of Computer Science, University of Maryland, April 1988.

[59] A. Wasserman, P. Pircher, and R. Muller, "The Object-Oriented Structured Design Notation for Software Design Representation," IEEE Software, pp. 50-63, March 1990.

[60] H. Tokuda and C. Mercer, "ARTS: A Distributed Real-Time Kernel," Operating Systems Review, vol. 23, no. 3, pp.29-53, July 1989.

[61] Y. Ishikawa, H. Tokuda, and C. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," ECOOP/OOPSLA '90 Proceedings, pp.289-298, October 21-25, 1990.

[62] D. Champeaux, L. Constantine, I. Jacobson, S. Mellor, P. Ward, and E. Yourdon, "Panel: Structured Analysis and Object-Oriented Analysis," ECOOP/OOPSLA '90 Proceedings, pp.135-139, October 21-25, 1990.

[63] P. Loy, "A Comparison of Object-Oriented and Structured Development Methods," ACM SIGSOFT: Software Engineering Notes, vol. 15, no. 1, pp.44-47, January, 1990.

[64] M. Schindler, Computer-Aided Software Design: Build Quality Software With CASE, New York, NY: Wiley, 1990.

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Disagree (Permission not granted)	Agree (Permission granted)
	Elmer Harold Moore
Student's signature	Student's signature
	11-29-90
Date	Date