A Survey on Integrating Object-Oriented Analysis and Formal Specifications

Wei Ding

March 31, 1998

1 Introduction

Formal specifications can be used to precisely describe the behavior of a computer system and its components. By using automated reasoning techniques, formal specification can also facilitate the development of a correct implementation.

OO lacks a simple model theoretical foundation for definition and discussion (unlike functional languages that can rely on the lambda calculus as basic formalism. The lambda calculus and its extensions can be used to develop better type systems for functional programming. We need formal models for the object-oriented programming paradigm. Most researchers in the area of object orientation acknowledge the need for formal methods in OO :

"The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundation for defining their semantics."

(Nierstrasz, 1992)

"Unlike functional programming, grounded in λ-calculus, or logic programming, grounded in logic, object oriented programming lacks a simple model that we can use as a basis for definition and discussion."

(Grogono & Gargul, 1994)

"The arguments over what is fundamental in object-oriented programming have existed for as long as the field. Which features - classes, prototypes, inheritance, delegation, message passing, encapsulation, abstraction - are at the heart of object - oriented programming, and how these things relate to one another, are not issues that will soon be resolved."

(Stein et. al., 1993)

A theoretical foundation for OO might help us in deciding which aspects are essential to OO and which are not (e.g. is the notion of a class essential, or can it be modeled using more primitive concepts?). It could also be used to give a formal semantics to concepts that are more or less understood, but have no generally accepted definition (e.g. What is an object? Should it incorporate state? Should it contain private methods?). A common framework is also useful for relating different approaches towards OO (e.g. class-based vs. object-based, delegation vs. inheritance, encapsulated inheritance vs. non-encapsulated inheritance). By expressing various object-oriented programming languages in the same formalism, one could compare their shortcomings and virtues, and eventually develop new languages with a more simple and orthogonal design, but with the same expressivity as currently available languages. Finally, formal methods could be used to establish properties of OOPL to find more efficient compilation techniques.

Layout of the Survey

The major goal of this survey is to describe the current state of the art in some formalisms in object-oriented analysis and design. First, we give a brief historical review of some earlier object-oriented analysis and design techniques. They are semi-formal or informal methods by themselves, but they form the basis of more formal techniques.

Then, we describe several extensions of Statecharts adopted for specifying behavior of OO programs: OO Statecharts, OMT and Objectcharts. Next, we discuss the most recent OO modeling language which are UML and ROOM. Finally, several popular tools that support OO analysis and design are introduced.

2 A Historical Review of Object-Oriented Analysis and Design Techniques

In this section, we give an overview of several semi-formal or informal object-oriented analysis and design techniques, which formed the basis of current more formal approaches.

2.1 State Transition Diagrams

State transition diagrams have been used right from the beginning in object-oriented modeling. The basic idea is to define a machine that has a number of states (hence the term finite state machine). The machine receives events from the outside world, and each event can cause the machine to transition from one state to another.

State transition diagrams were around long before object modeling. They give an explicit, even a formal definition of behavior. The big disadvantage is that they force the designers to define the possible states of a system. In a large system, the growth in the number of states is exponential. Object-oriented methods define separate state-transition diagrams for each class.

Instead of a "flat" FSM, they adopt a hierarchical FSM, and this somehow eliminates the explosion problem, since each class is simple enough to have a comprehensible state transition diagram.

The drawing of *State transition diagrams* consists of ellipses representing states and these are joined by directed arcs which indicate the transitions between states. The state ellipses are annotated with the state name. The transition arcs are annotated by the event that triggers the transition and can also have an annotation indicating an event or function that is triggered by the transition. Figure 1 gives an example of a Link Management State transition diagram. The *DL.CONNECT.REQUEST* is the event that triggers a transition to a *SETUP PENDING* state and during the transition, a *SETUP* frame is sent to the destination *DCE.*

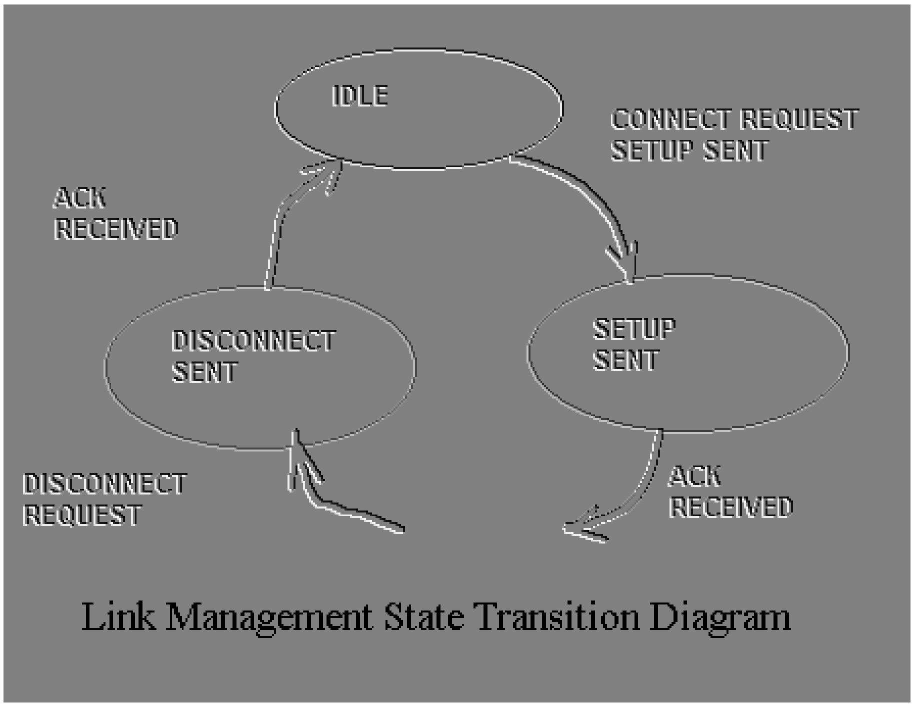

Figure 1: A Link Management State Transition Diagram

How and When to Use Them Effectively

State models are ideal for describing behavior of a single object. Their formality allows building of tools which can execute them. One might find their biggest limitation is that they are not good at describing behavior of several objects. However, combining finite-state machine with interaction diagram (introduced in the next section) can specify those cases more effectively.

2.2 Use Cases

Object developers often assume the existence of some requirements document. Such a document models the information for their modeling and development, but what should it look like is unclear. Jacobson [9] pushed the idea of use-cases to the center of the stage.

A *use-cases* occurs when a user performs "a behaviorally related sequence of transactions in a dialogue with the system" [9]. *Use-case diagrams* provide a way of describing the external view of the system and its interactions with the outside world . The outside world is represented by actors, which are roles played by various people, or other computer systems. One person may play many roles, and a role may be played by many people. Use cases represent an informal, high-level view of the whole system. This approach is not designed to be complete; instead, use cases are used when needed to describe general aspects of the system.

Figure 2 gives an example of a web site Use-Case diagram. *Customer* and *Account Agency* are the actors in the system. The line with solid arrows represents actor accessing to some special use case. The line with non-solid arrows represents one uses case is using another use case. In this diagram, *Customer* accesses uses a case *View Order;* use case *Shopping* uses another use case *Browsing* and *Ordering.* (Note: we also put the label *uses* to indicate that one use case is using another one. If we put the *extends* instead, it means that the Uses case is inheriting another use case.)

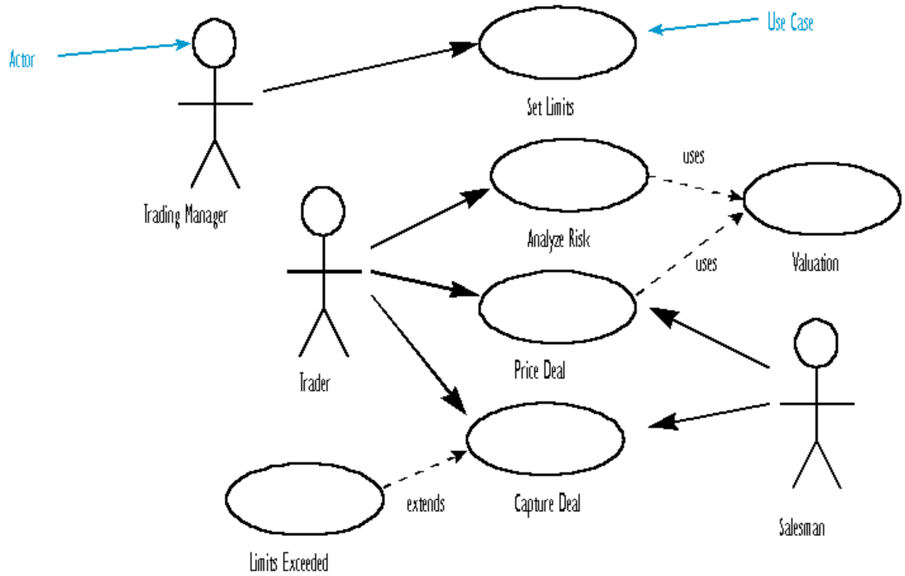

Figure 2: A Use-Case Diagram

How and When to Use them Effectively

Use cases are vital part of OO development. Whenever there is a need to understand the requirements of the system, use cases should be used. Whether or not to use the diagrams is another matter, and it really depends on the usage of it. Use cases are valuable if just kept on a database as an unstructured list. Each use-case needs a name and a few paragraphs of description. They are central to planning the evolutionary development process. They should also drive system testing.

2.3 Interaction Diagrams

Interaction diagrams [11] are models that describe how a group of objects collaborate in some behavior - typically a single use-case. The diagrams show a number of example objects and the messages that are passed between these objects within a use-case.

Interaction diagrams come in two forms: *sequence diagrams* [9] and *collaboration diagrams.* In *sequence diagram,*objects are shown as vertical lines with messages as horizontal lines between them. Figure 3 shows an example of sequence diagram. In this diagram, the *Order entry window* sends a *prepare* message to an *order,* and the *Order* outputs a prepare message, **prepare* indicates that *prepare* is a output message here. The *order line* first checks the stock item, and if the check returns true, it removes stock from the stock item. If stock item falls below the reorder level, it requests a new delivery.

The other form of the interaction diagram is the *collaboration diagram,* as shown in Figure 4. In this diagram, the example objects are shown as icons. Arrows indicate messages sent in the use case. This time, the sequence is indicated by a numbering scheme. Simple collaboration diagrams just number the messages in sequence. More complex schemes use a decimal numbering approach (e.g. 2.3) to indicate if messages are sent as part of the implementation of another message.

The major difference between *sequence* and *collaboration diagrams* is that the former is organized by time, while the later one is organized by space.

How and When to Use Them Effectively

Interaction diagrams are most useful when we want a view of the behavior of several objects within a single use case. They are good at showing collaborations between the objects, and provide a scenario based specification useful for many developers. However, they are not very good at precise definition of the behavior. If the behavior of a single object across many use-cases is wanted, a state transition diagram might be a better choice.

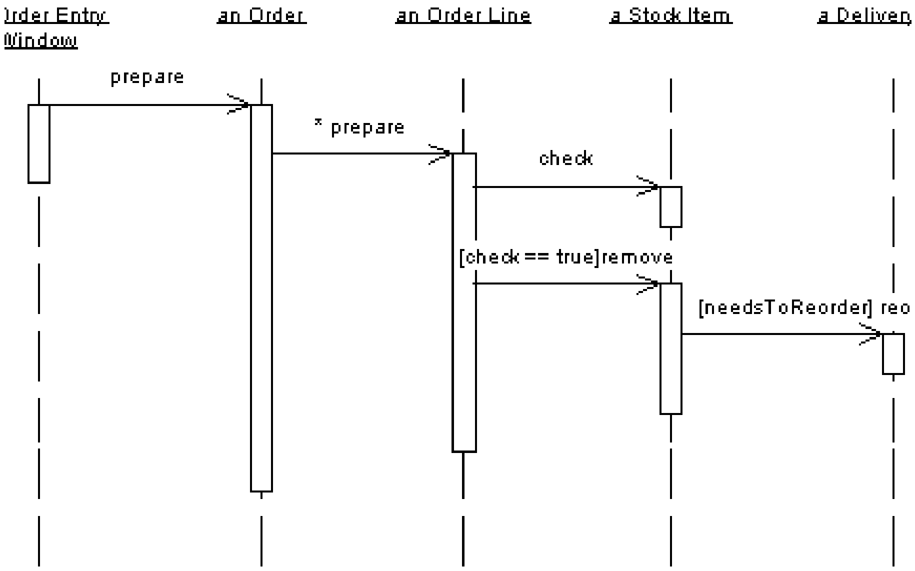

Figure 3: A Sequence Diagram

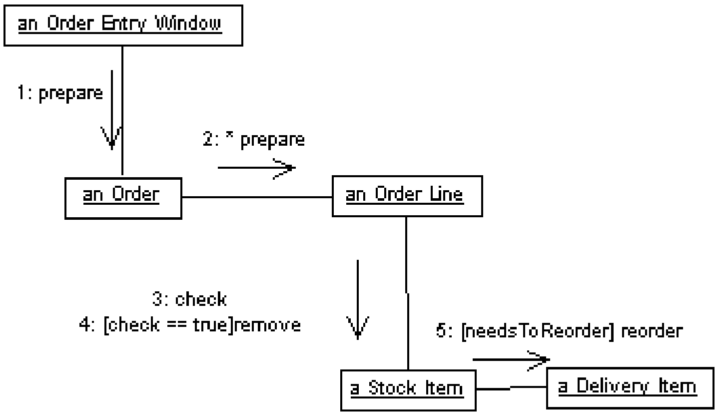

Figure 4: A collaboration Diagram

2.4 Class Diagrams

Class diagrams [3] is a central modeling technique that runs through nearly all object-oriented methods. Class diagrams describe types of objects in the system and various kinds of static relationships between them. There are three principal kinds of relationships which are important: associations (e.g. a customer can rent a number of videos), subtypes (e.g. a teacher is a kind of person) and aggregation (e.g. an engine is part of an aircraft).

Associations represent relationships between instances of types (a person works for a company; a company has a number of offices). A distinction is made between attributes and associations. Attributes are seen as internal to a class while associations link classes externally. Also, attributes refer to values which are non-objects. Figure 5 gives an example of a class diagram for an ordering line. In this Figure, *Product* and *Quantity* are attributes of the Order Line class; while *Order class* is associated with *Customer class.* Associations are allowed to be subtyped as types which define how subtypes can place further constraints on inherited associations.

Aggregation relationships indicate part/whole relationships (a hammer is made up of a head and a raft). Again in the Figure 5, Order Line is part of the *Order,* so relationship between them is aggregation.

Subtyping is modeling equivalent to inheritance in OO programming. As shown in the diagram, *personal customer* and *corporate customer* are both customers. They have differences as well as similarities. For example, in Figure 5 , the similarities can be placed in a general *Customer* class with *Personal* and *Corporate Customer* as subtypes.

A type usually includes two parts: attributes and the *operations.* These operations are the only methods which can be operated on the attributes.

Perspectives

Cook and Daniels [3] suggested three perspectives that users of class diagrams might want: conceptual, specification and implementation diagrams.

- *Conceptual:* In this case, the diagram represents the concepts in the domain under study. Indeed the model is drawn with little or no regard for the software that might implement it, and is generally language independent.
- *Specification:* At this stage, the software is being considered, but only the interface of the software, not the implementation yet. We are thus looking at types (which are indeed the abstraction of the classes) rather than classes.
- *Implementation:* In this view, classes and implementation details are considered.

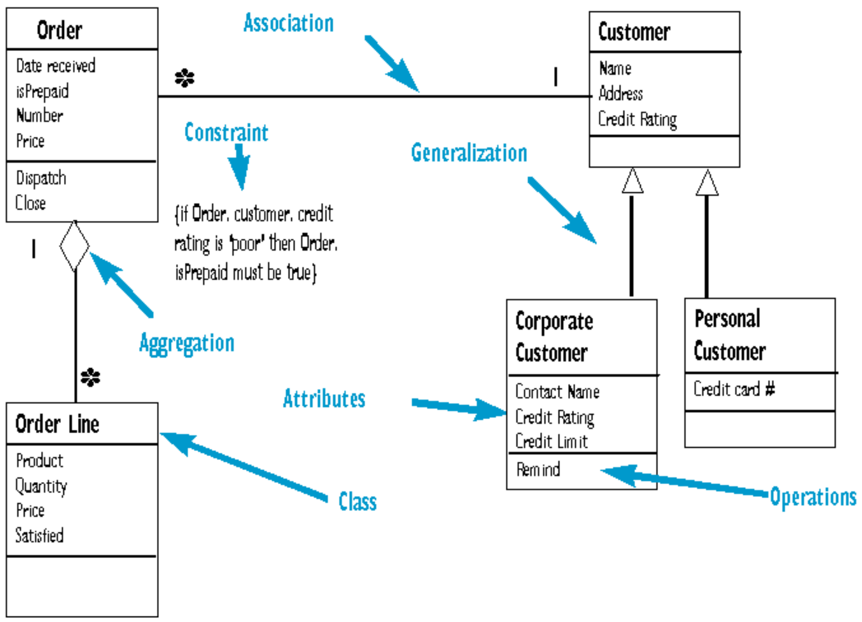

Figure 5: A Class Diagram

How and When to Use them Effectively

Class diagrams are the backbone of nearly all OO methods, so they are being used very often. The trouble is that they are so rich that they can be overwhelming to use. Here are some techniques that are considered to be useful for class diagrams users:

- Start with the simple stuff: classes, associations, attributes, and generalization. Other notations (such as constraint, operations, etc) should be introduced only when it is really necessary;
- Sort out which perspective are being drawn the models from. If it is in analysis, draw conceptual models. When working with software, concentrate on specification models. Draw implementation models only when illustrating a particular implementation technique;
- Generally, not everything needs to be modeled. A lot of the time, it is helpful to only concentrate on the key areas. It is better to have a few diagrams that are used and keep up to date than many forgotten, out-of-date models.

2.5 Package Diagrams

Sometime we want to draw a class model for a large system, and it is too big to comprehend. We could just chop the diagram into several pages, but while this makes the diagram easier to understand, the underlying software is just as likely to have lots of links that make the code difficult to understand and fragile.

A useful technique to handle this is that of *packages,* based on Booch's[1] class categories. *Package Diagram* is not a separate technique; the purpose of this approach is to create a higher-level architectural view of the class interactions. In this technique, each class is put into a single package. A class can use classes in the same package freely. However, if a class wants to use a class in a different package, it must draw a dependency (called by Booch a *visibility*) to that package. The overall picture of the system is the picture of packages and their dependencies, where the aim is to minimize the number of dependencies.

Each class inside the package can be either public or private. Public classes can be used by all packages that have visibility, while private can only be used by classes within the same package. A little icon above the class box represents a package. The dashed line represents a dependency relationship. Figure 6 gives an example of a Package Diagram for a GUI Library system. In this Figure, a package *Portfolio Application* wants to use a class inside the package *Scenario Manager,* thus drawing dependency to the *Scenario Manager* is required.

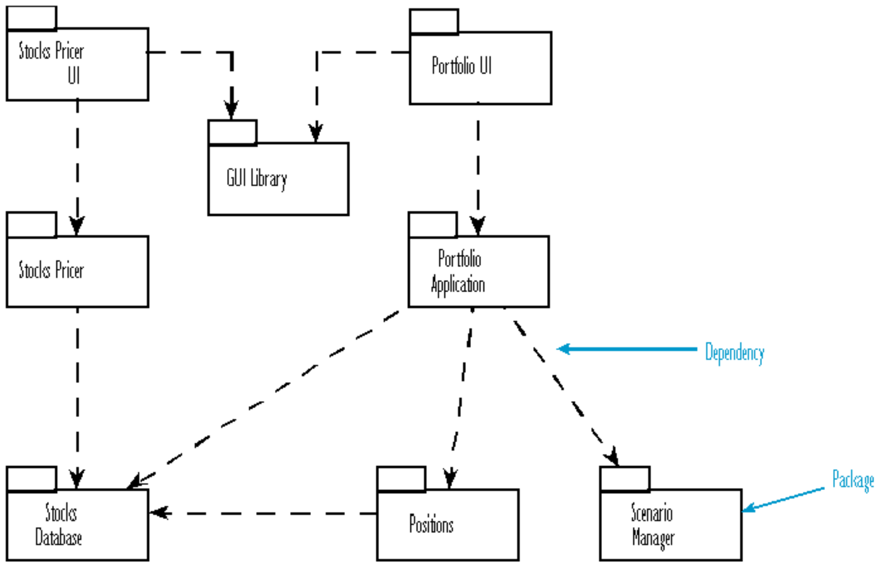

Figure 6: A Package Diagram

How and When to Use Them Effectively

Packages are a vital technique for any large scale system. Both the UML [16] and Booch [1] do not treat package diagrams as a separate technique. They are more useful in large scale system.

2.6 Statecharts

Statecharts describe two aspects of object behaviors: how objects communicate and collaborate and how they carry out their own internal behavior. Important features of OO such as inheritance and polymorphism can be easily modeled in Statecharts.

The semantics we use for OO-Statecharts is close to what has been defined in Statemate [5]. Reactions to events are step by step: the events and actions generated in one transition do not take effect until the next step, after a stable state has been reached. The difference between the Statecharts we are introducing here and Statemate's approach is that in Statemate, all triggers are constantly attentive, and generated events reach their destinations instantly; but here, events are handed to server objects one by one from the queue, in single-event processing. Thus, the new approach is aimed at achieving a more realistic design and implementation environment.

Statecharts involve reactions of the form:

trigger[condition] $\backslash$ *action-list* ,

all parts of which are optional. Such a reaction can adorn a transition arrow as on the bottom of Figure 7. Figure 7 gives the main statechart for a rail-car system. The @ prefix denotes the presence of a more detailed statechart. A car can be in one of the five states: *idle, standby, arrival, cruising or departure.* All arrows and expressions beside the arrows are reactions which trigger transitions between states of an object. In the top-level statechart of Car system, *new* is a trigger creates new instances *term.*

3 OO-Extensions of Modeling Techniques

3.1 Statecharts for Object-Models

Most object-oriented modeling techniques are not behaviorally expressive and rigorous enough to permit production of fully executable models and the automatic synthesis of efficient code in languages such as C++. These artifacts can only be developed from a formally specified model. Thus, various researchers set out to develop such formalisms. The one that has been used extensively is an OO extension of Statecharts.

This technique consists of two parts [7] :

- " *Object-model diagrams* specify system structure by identifying object classes and their multiplicities, object relationships and roles, and subclassing relationships.".
- " *Statecharts* describe system behavior. A Statechart attached to a class specifies all behavioral aspects of the objects in that class.".

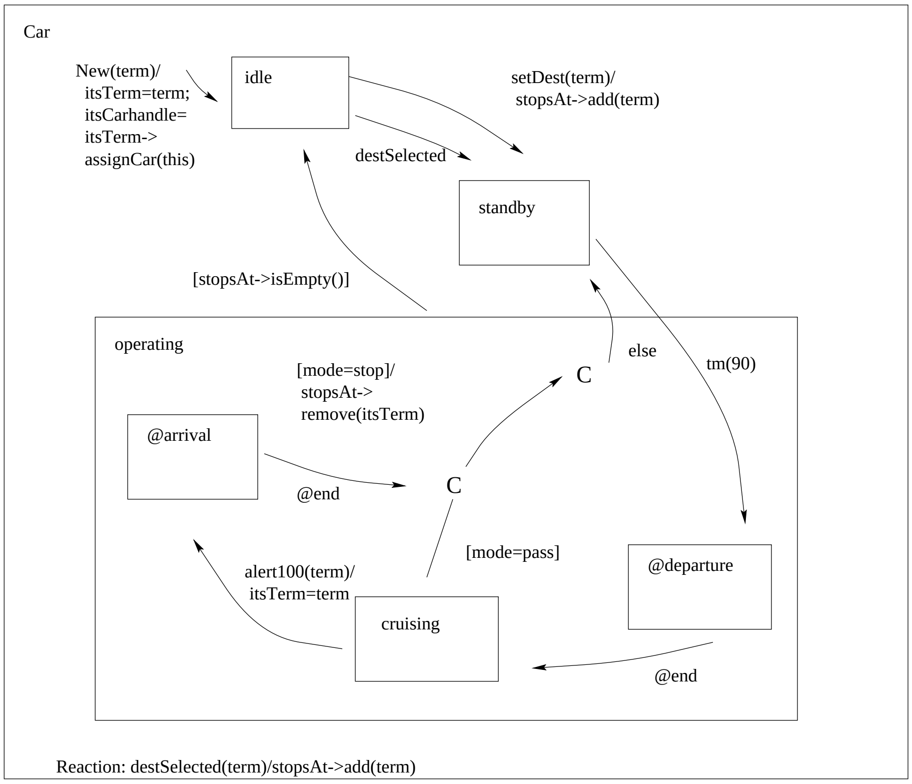

Figure 7: Top-level Statechart of Car

The Statecharts were introduced in Section 2.6, so here we just present the description of object-model diagrams.

Object-Model Diagrams

An *object-model diagram* specifies classes of a system and their structural relationships. It enables representation of static and dynamic aspects of the system. It is similar to an entity-relationship diagram, with directed edges representing relationships and undirected edges acting same as two-way directed edges. Figure 8 shows a partial object-model diagram for the railcar system with four main classes, two of which, *Car* and *Terminal,* are composites. The numbers in the boxes indicate multiplicity information (instances): one *ControlCenter* and six *Terminals.* The other classes contain an asterisk, which means they can have unlimited instances. The figure also shows four many-to-one bidirectional association relationships and two unidirectional ones.

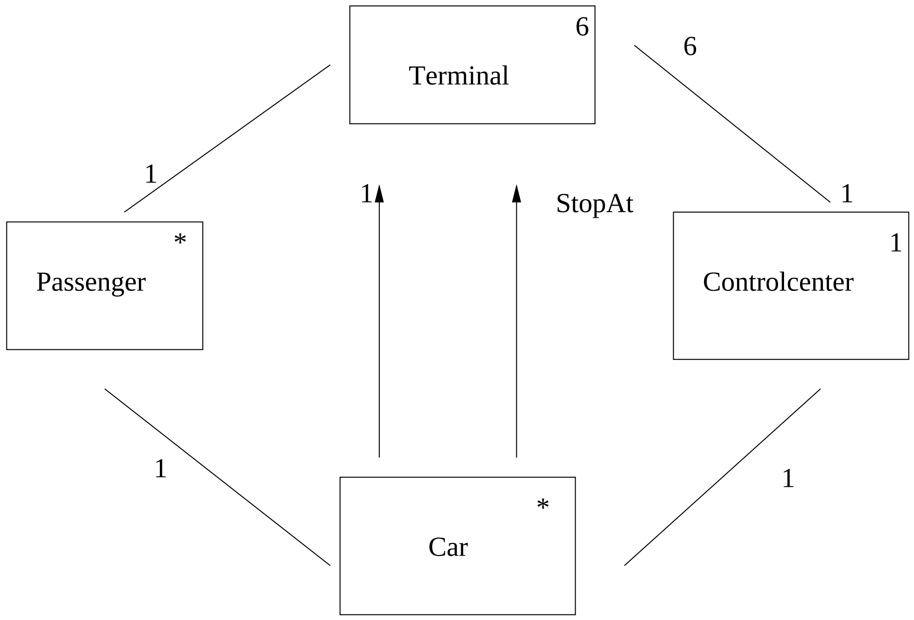

Figure 8: High-level object-model diagram for the railcar system

3.2 Objectcharts

Objectcharts [8] is an extension to Statecharts. This notation is for specifying the behavior of object classes as state machines [4] . The approach combines object-oriented analysis and design techniques and Harel's Statecharts [7] such that it offers a formal and declarative extension to state machines.

Informally, an *Objectchart* specification of an object class can be defined as a Statechart together with a set of *transition* specifications and a set of *invariant* specifications. In order to explain what are transition specifications and invariant specifications, we need to know what *service* and *observers* are. A *service* is an abstraction of a method in object- oriented programming.

Some services don't change the state of an object - some just report on the value of attributes. These services are called *observers.*

A *transition specification* comprises the initial and final state names of the transition and the service name for the transition, together with a firing condition and a postcondition. A transition should be given for each arc in the Statechart. A firing condition describes restrictions on a transition imposed in terms of the attributes and observers of the class. A postcondition for a transition is a predicate on the initial and final values of attributes and observers which characterizes the effect of the transition. Some observers or attributes of a class may be redundant, such as they can be derived from others. The derived observers and attributes can be specified as *invariant* relations on the observers and attributes used to annotate the states. An *invariant specification* comprises a state name and the relation which holds in that state.

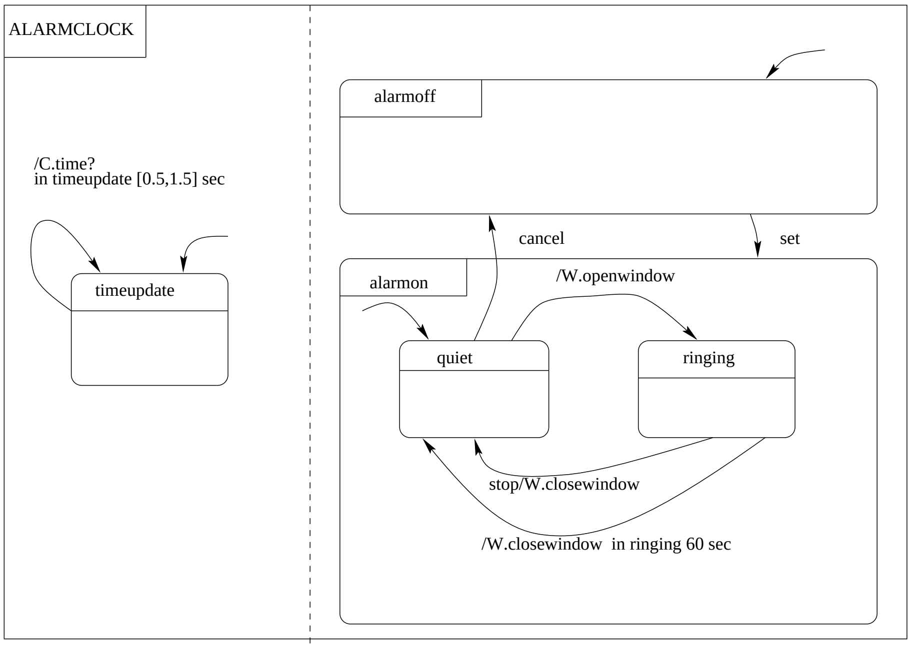

Figure 9: Alarm-clock Statechart

Figure 9 gives an example of an Objectchart via an alarm clock system. In this example, the provided services *set* and *cancel* can change the state of an alarm clock. The alarm is initially off and can be set. Once set, it may be cancelled. The transition on the timeupdate state indicates that the required service *C.time?* is requested approximately every second. Notice that the prefix

‘ \ ’ is just a reminder that *C.time* is a required service. Then the alarm-on state is refined into two states: *quiet* and *ringing.* In order to move from quiet to ringing the *openwindow* service must be requested from some object *W.* In the configuration of this example, *W* is the bell Window. In the ringing state the window is closed after a fixed duration or when the user stops it. The alarm must be in the quiet state before an alarm setting can be cancelled.

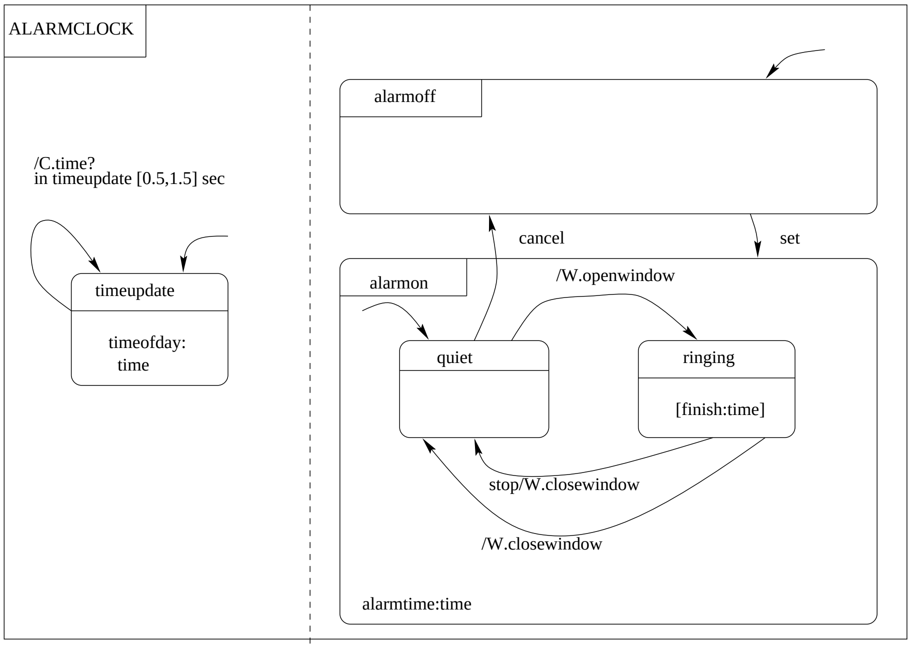

Figure 10: Alarm-clock-enhanced Statechart

Figure 10 shows the enhanced Statechart for the alarm clock system. The alarm clock uses the *C.time?* to define the *timeofday.* The attribute *alarmtime* is only visible when the alarm is set. The timed transition from ringing to quiet is replaced by introducing a hidden attribute, *finish,* which characterizes how long the bell window is open before it closes itself. The brackets around *finish* indicate that it is not visible outside the alarm clock.

One of the transition specification of this alarm clock system is :

timeupdate - $\rangle$ timeupdate: $\{$true$\}$ /*C.time?*($\|$t) $\{$timeofday = t$\}$

timeupdate are both the initial and final state names of the transition, and /C.time? is the service name together with *true* as firing condition and *timeofday* $= t$ as postcondition. Notice that some necessary parameters are added to the services. Parameters are partitioned into input and output by a “$\|$”. Here, *C.time?* ($\|$t) has an output parameter t , and the value of t is set by the object C that provides the service *time* . Hence this specification means, starting from the initial state called *timeupdate,* provided that *true* is satisfied, *timeofday* $= t$ will be satisfied after the transition, service *C.time?* outputs *t,* and the final state is *timeupdate.*

An example of an *invariant specification* is :

quiet:

{timetoalarm = alarmtime - timeofday}

with *quiet* as a state name and *timetoalarm = alarmtime - timeofday* as a relation which holds in that state.

Relationship between Harel's Statecharts, Statecharts for Object-Models and Objectcharts

Both Statecharts for Object-Models and Objectcharts are enhanced Harel's Statecharts. Statecharts for Object-Models are Statecharts with Object-model diagrams. These diagrams specify static and dynamic aspects of the system structure. But Statecharts for Object-models describe system's behavior the same as Harel's Statecharts.

Objectcharts extends Harel's Statecharts in the way of specifying the behavior of object classes. There are some subtle but important changes from Harel's original Statecharts to Objectcharts. Harel's broadcast communication primitive is not used in Objectcharts, since it does not provide a suitable model for the object communication. Instead, the new approach uses a simple primitive in which service request and complete service execution are treated as a single atomic event.

3.3 The Object Modeling Technique

Object Modeling Technique (OMT), developed by J. Rumbaugh [15] , can be seen as a a design methodology which consists of a collection of predefined techniques and notational conventions in order to support the process, which consists of several phases (e.g. analysis, design and implementation) for the organized production of software.

The OMT methodology uses three kinds of models to describe a system, each capturing aspects of the system, but all required for a complete description. The *object model* represents the static, structural, "data" aspects of a system. The *dynamic model* represents the temporal, behavioral, "control" aspects of a system. The *functional model* represents the transformational, "function" aspects of a system. A typical software procedure incorporates all three aspects: it uses data structures (object model), sequences operations in time (dynamic model), and transforms values (functional model). Each model contains references to entities in other models. For example, operations are attached to objects in the object model but more fully expanded in the functional model.

The three kinds of models separate a system into orthogonal views that can be represented and manipulated using a uniform notation. The different models are not completely independent – a system is more than a collection of independent parts – but each model can be examined and understood by itself to a large extent. The interconnections between the different models are limited and explicit. Of course, it is always possible to create bad designs in which the three models are so intertwined that they cannot be separated, but a good design isolates the different aspects of a system and limits the coupling between them.

3.3.1 OMT Object Model

OMT Object model [15] describes the static structure of its objects, classes, and their relationships to each another. It is the most fundamental one among the three models, because it provides the essential framework into which the dynamic and functional models can be placed. The content of an object model is decided by its relevance to an application. An object class describes a group of objects with common attributes, operations and semantics.

Links and associations establish relationships among objects and classes. A link connects two or more objects. An association describes a group of links with common structure and common semantics. Aggregation is a tightly coupled form of associations with special semantics, which means an aggregate object is made of components. Components are part of the aggregate.

The various object modeling constructs work together to describe a complex system precisely. Once an object model is available, the model can be compared against the knowledge of the real world or the desired application. In Figure 11, class *Shape* defines common parameters of all kinds of shapes, including a rectangular boundary defined by the attributes *color, line-width* and operations. *Line* and *ClosedShape* are varieties of shapes. A polygon consists of an ordered list of vertices, shown as an aggregation of many points.

3.3.2 OMT Dynamic Model

The dynamic model of OMT is based upon Harel Statecharts [6]. This model represents control information: the sequences of events, states, and operations that occur within the system of objects. The dynamic model controls which operations are performed and the order in which they are applied.

3.3.3 OMT Functional Model

Functional models [15] in OMT specify how operations derive output values from their input values, without regard to the order of computation. They consist of data flow diagrams (DFD) with directed named edges linking process nodes to process nodes or linking process and data nodes. The names of edges describe the data which is passed in or out of each process. Two forms of data are distinguished: *actor* objects which produce and consume data, and *data store* objects which store data passively. The process model notation is not mathematically precise; it

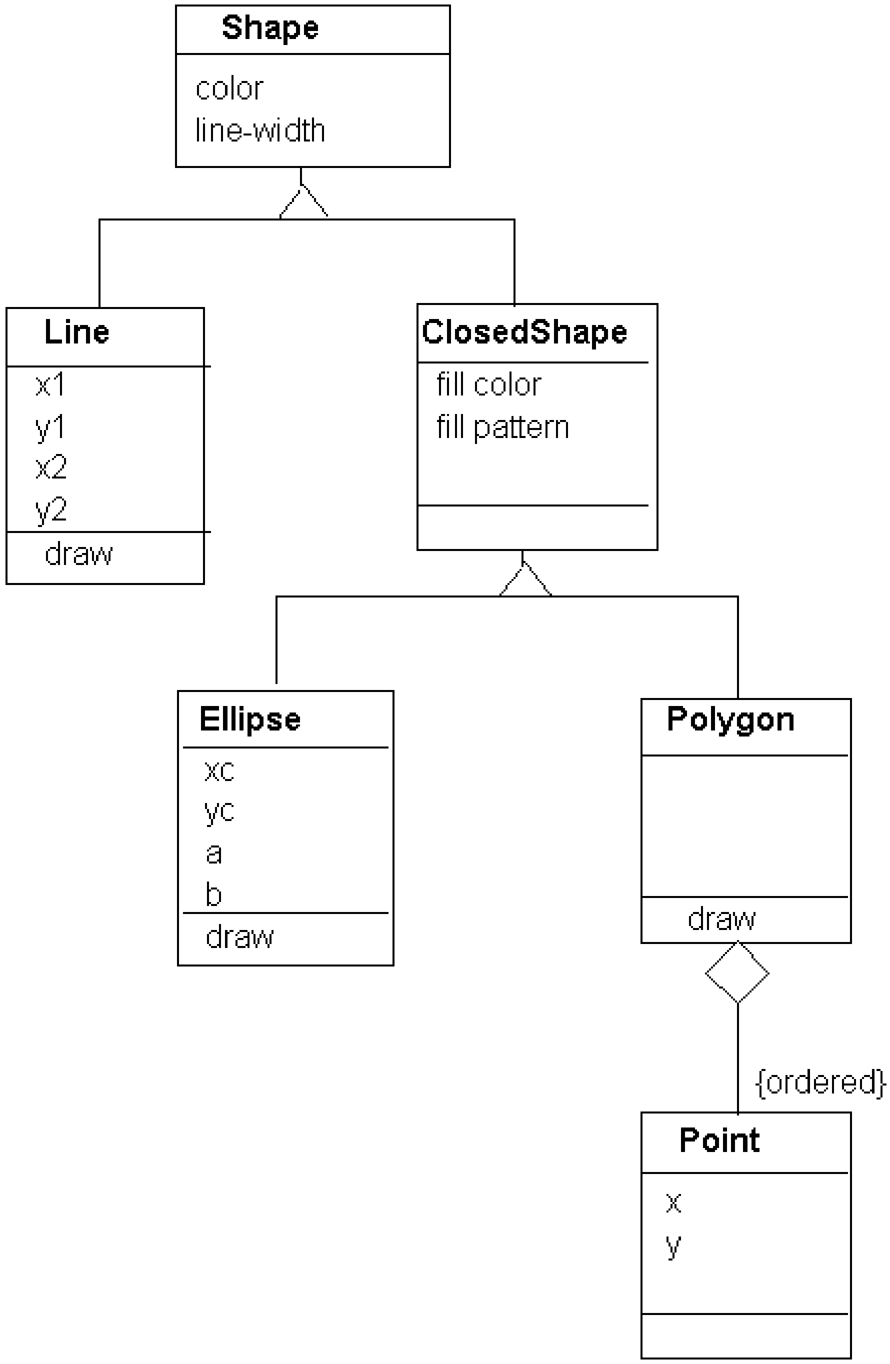

Figure 11: OMT Object Model

can only be used to outline method definitions and to provide a view of the usage dependencies between methods. "It is the least well-developed model in OMT" [12] .

Figure 12 gives an example of a data flow diagram. As this Figure illustrates, a data flow diagram contains *processes* that transform data, *data flows* that move data, *actor* objects that produce and consume data, and *data store* objects that store data passively. All processes are drawn as ellipses containing a description of the transformation, usually its name. So *Get Diagram Info., Get Objects on Page, Create Postscript Language* and *Display EPS File* are processes. Each process has a fixed number of input and output data arrows. A data flow connects the output of an object or process to the input of another object or process. It represents an intermediate data value within a computation. For example, in Figure 12, User produces *preview request* to the *Get Diagram Info.* process, which produces *view context* request to the *Get Objects on Page* process, etc.

An *actor* is an active object that drives the data flow graph by producing or consuming values. An actor is drawn as a rectangle (e.g. *User, Postscript Previewer, Window*) to show that it is an object. Arrows between the actor and the diagram are inputs and outputs of the diagram. A *data store* is a passive object within a data flow diagram that stores data for later access. A data store is drawn as a pair of parallel lines containing the name of the store (e.g. *System Database, Encapsulated Postscript File*). *System Database* and *Encapsulated Postscript File* merely respond to requests to store and access data instead of generating any operations on their own. Input arrows indicate information or operations that modify the stored data, such as *eps text* ; output arrows indicate information retrieved from the store, such as *current view* .

3.3.4 Attempting to Formalize Object Model Diagrams

Although OMT provides the user with easy to understand graphical notations for expressing a wide variety of concepts central to the presentation of software requirements, it lacks of formality, and thus it cannot be evaluated of completeness, consistency, and content in requirements and design specifications [2] .

Rumbaugh, *et al.* [15] defined the semantics of object model in part, by displaying a series of *instance diagrams* as examples of what a specific object model describes. An *instance diagram* describes how a particular set of objects relate to each other, and it describes object instances. The algebraic formalization of the semantics for the object model and instance diagram notations are introduced below. The key concepts of this formalization process are highlighted in Figure 13. The arrow labeled "OMT semantics" represents the relationship between object models and instance diagrams that has been described by Rumbaugh. The new approach proposes that instance diagrams can be used to provide a graphical definition of semantics for object models. In this case, object models are formalized as algebraic specifications, and instance diagrams are

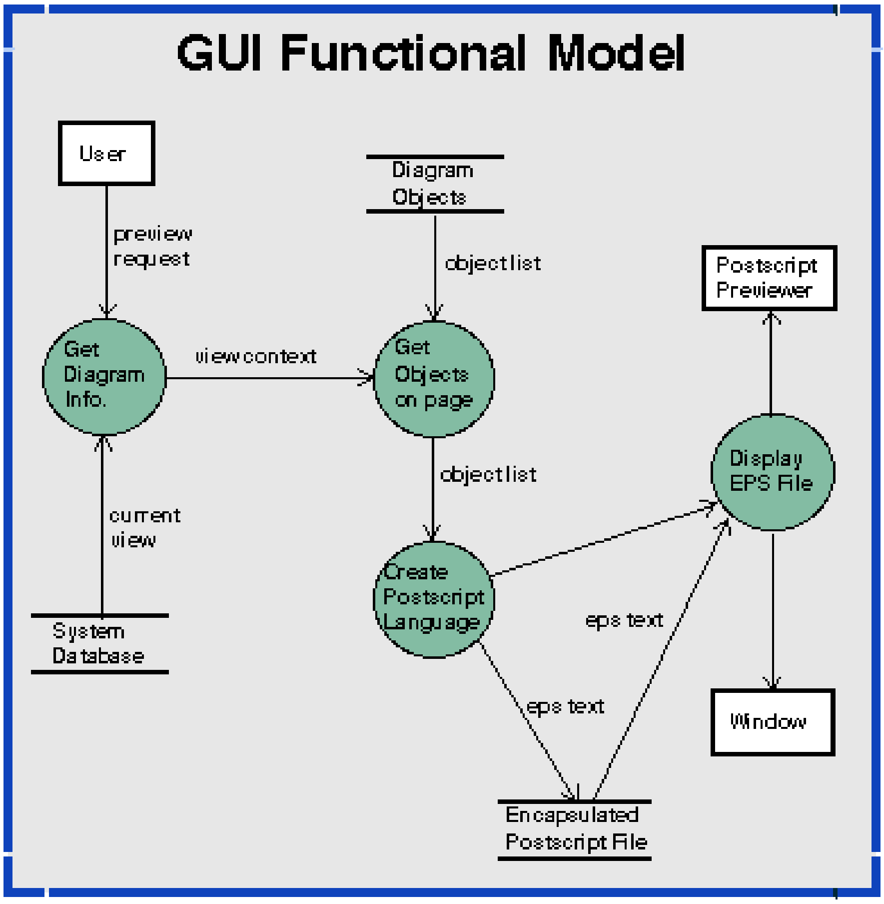

Figure 12: OMT Functional Model

formalized as algebras.

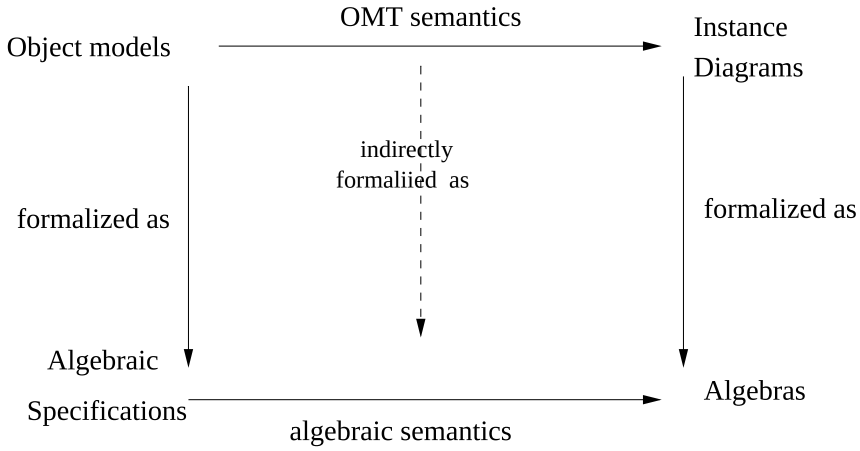

Figure 13: Basic approach to formalization

The formal semantics for the OMT object model notation [2] enables object model diagrams to be treated as formal specifications, and thereby overcomes many of the analytical limitations in this otherwise informal method. The formalization method describes a technique for deriving modular algebraic specifications directly form object model diagrams.

Some extensions to the OMT object model notation have been made in the formal method, such as graph-based notations for attributes and object states. The notation of external classes has also been introduced to allow for the incorporation of algebraic specifications that have no graphical specification.

4 Third - Generation Modeling Techniques

Both UML and ROOM are third-generation modeling methodologies for analyzing and specifying object-oriented systems.

4.1 The Unified Modeling Language

The Unified Modeling Language (UML) [16] is a third-generation method for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. The first-generation specifying method is called object-oriented analysis and design (OOA & D), and the second-generation specifying method is by Booch and Rumbaugh (e.g. OMT). UML represents the unification of the Booch[1], and OMT methods, and additionally incorporates ideas from

[1]The BOOCH method is a widely used OO - method, that helps designers design the systems using the object paradigm

a number of other methodologists.

UML is meant to be applicable to the modeling of all types of systems. It applies equally well to real-time systems, client/server, and other kinds of "standard" software applications. It provides a rich set of notations and promises to be supported by all major CASE tool vendors. The primary goals of the UML are:

- Give developers a ready-to-use, expressive visual modeling language so that they can develop and exchange meaningful models;
- Provide extensibility and specialization mechanisms to extend the core concepts;
- Be independent of particular programming languages and development processes;
- Provide a formal basis for understanding the modeling language;
- Support higher-level development concepts such as collaborations, frameworks, patterns, and components;
- Integrate best practices.

The UML distinguishes between the notions of model and diagram. A *model* contains all of the underlying elements of information about a system under consideration and does so independently of how those elements are visually presented. A *diagram* is a particular visualization of certain kinds of elements from a model, and generally exposes only a subset of their elements' detailed information. A given model element might exist on multiple diagrams, but there is but one definition of that element in the underlying model. The diagrams supported in the UML include :

- Class diagram
- Use-case diagram
- Interaction diagram
  - Sequence diagram
  - Collaboration diagram
- State diagram
- Component diagram [2]

[2] This diagram illustrates dependencies among software components, including source code components, binary code components, and executable components.

- Deployment diagram [3]

The UML defines a graphical language for describing, or modeling, the various components of a software system. A standard modeling language makes it easier for developers to incorporate modeling techniques into application development projects.

4.2 ROOM

The real-time Object-Oriented modeling (ROOM) methodology is a practical method that blends the advantages of the object paradigm with advanced modeling concepts expressly designed for developing real-time software. It places special emphasis on the higher "architectural" levels that are keys to constructing reliable comprehensible, and evolvable software. ROOM is also characterized by its capacity to exploit computer-based automation (through executable models, component reuse, and automatic code generation techniques) to enhance quality and productivity.

ROOM is similar to other popular OO modeling methodologies, and in fact, a future ObjecTime will support UML-RT, which is an extension of the standard UML notation designed to address the challenges of real-time systems. Like most OO modeling methodologies, ROOM defines object models that express the compile-time relationships between objects, state diagrams to specify object behavior, and message sequence charts to illustrate the run-time interactions between objects.

5 Object-Oriented Design Tools

Several commercial tools that support OO analysis and design are available. This section summarizes some of the more popular ones.

5.1 i-Logix Rhapsody

i-Logix' Rhapsody [10] is an object-oriented analysis and design tool. Using Rhapsody, software developers can graphically design the software components of a system using object-oriented techniques. Rhapsody generates complete production-quality code from the design diagrams and enables design-level debugging and verification. Once an object model has been built with the designer, production-ready code can be generated automatically.

Rhapsody is the only object-oriented tool based on the UML (Unified Modeling Language) standard capable of generating complete production quality code from design diagrams, and then animating execution behavior. Rhapsody is enhanced to allow embedded system software developers to share object-oriented design information in a work-group environment.

[3] This diagram illustrates the configuration of run-time processing elements and the software components, processes, and objects that live on the processes.

Rhapsody's Designer enables composition of three types of diagrams : Message Sequence Charts [4], Object Model Diagrams and Statecharts. Each diagram allows the user to focus on a different aspect of the model, simplifying the design process.

5.2 Rational Rose

Rational Rose [14] is a graphical component modeling and development tool that enables organizations both to model software applications that meet current business needs and to evolve them easily as new requirements emerge. It allows developers to step back and graphically visualize their applications using the Booch method, the Object Modeling Technique, and the industry-standard notation, the Unified Modeling Language (UML).

Rational Rose provides the visual modeling environment of creation and documentation of use cases; drawing the Use Case diagram, dragging actors, and use cases on to the diagram from the browser; creating of a sequence diagram and modeling in it; creating classes in the model using the browser; adding attributes to a class; adding behavior/operations to a class; adding associations between classes; modeling the inherit relationship on a class diagram; creation of transitions between various states on a State Transition Diagram, creation different actions in a state diagram; modeling using the component diagram; modeling using the deployment diagram.

5.3 ObjecTime

ObjecTime [13] is the visual development environment to deliver production code from graphical designs. It supports and services for complex, event-driven applications. ObjecTime is a CASE tool that supplies a powerful graphical modeling environment for the object-oriented design and simulation of event-driven real-time systems. This toolset uses the real-time Object-Oriented Modeling (ROOM) methodology.

ObjecTime supplies a language derived from Smalltalk which supports rapid prototyping of high-level design models. This language is called the Rapid prototyping Language (RPL). Like Smalltalk, RPL is also object oriented, typeless, and interpreted. Unlike Smalltalk, RPL relies on ALGOL-style control structures (such as *for/next, do/while,* and so on), which are more familiar to most real-time software designers.

Figure 14 illustrates the RPL syntax for performing a typical task in ObjecTime: sending a message to an object, and storing the response in a local variable.

[4] Message Sequence Charts is another name of Sequence Diagrams

```
init: infoIn
| msgReturn |
msgReturn :=
   INVOKE unitData
      SIGNAL %parameterRequest
      DATA infoIn
   ENDINVOKE.
startupParameters := msgReturn data.^nil
```

Figure 14: RPL syntax for performing a typical task in ObjecTime

5.4 UBET

UBET is a tool to help developers to specify the behavior of distributed systems. It is a concrete realization of a vision for how behavioral requirements of systems can be captured efficiently and effectively. The main features of UBET are:

- UBET supports capture of Use Cases in a structured and concise way. Hierarchical Use Cases are used to represent behaviors in UBET, and it lets the users to graphically define Use Cases as diagrams, allowing exception handling to be specified easily.
- UBET supports an interface with the ObjecTime tool for import of object-oriented models, that can be used to constrain the naming of system components defined in UBET.

The current enhancements being worked for UBET are: automatic test case selection and generation, and extraction of initial finite state machine models for use in verification or to jumpstart development models.

5.5 Software through Pictures

Software through Pictures (StP) provides the designers an integrated multi-user environment sharing a common architecture and central repository. At the first step, designers chose the modeling method which fits their best needs. The method choices include:

- Object-Oriented methods include:
  - Unified Modeling Language
  - Object Modeling Technique
  - Booch
- Structured Environments methods

- Information Modeling methods

After the designers have gained better understanding of their needs through modeling, StP produces test cases and automatically extracts test specifications from project models.

6 Conclusion

In this survey, We looked at the underlying techniques that each semi-formal or formal OO-specification method used, and highlighting their differences and similarities. Among these methods, I believe OMT and UML will become the dominant object modeling methods. However, there are still many useful techniques that are either not present in the OMT & UML, or not stressed enough. Also many of the techniques in UML have not been written up in a appropriate tutorial form. All these things are need to be done in the near future.

References

[1] G Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, Menlo Park, CA, second edition, 1993.

[2] Robert H. Bourdeau and Betty H. C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions of Software Engineering*, October 1995.

[3] S Cook and J Daniel. *Designing Object Systems: object-oriented modeling with syntropy.* Prentice Hall International, Hemel Hempstead, UK, 1994.

[4] P. J. Denning, J. B Dennis, and J. E. Qualitz. *Mahines, Languages and Computation.* NJ: Prentice-Hall, Englewood Cliffs, 1978.

[5] D.Harel and A.Naamad. The statemate semantics of statecharts. *ACM Trans. Software Eng. Methodology*, Oct. 1996.

[6] D Harel. *Statecharts: A visual formalism for complex systems.* Science of Computer Programming, Springer-Verlag, 8 edition, 1987.

[7] David Harel and Eran Gery. Executable Object Modeling with Statecharts. *Computer*, pages 31–42, July 1997.

[8] Fiona Hayes, Derek Coleman, and S. Bear. Introducing objectcharts or how to use statecharts in object oriented design. Technical Memo ?, Information Management Lab, Hewlett Packard Laboratories, November 1989.

[9] Jacobson I, Christerson M, Jonsson P., and Overgaard G. *Object-Oriented Software Engineering: a use case driven approach.* Addison-Wesley, 1992.

[10] the i-logix. Technical report, i-Logix Inc., Three Riverside Drive, Andover, MA 01810.

[11] Tomonari Kanba and Osamu Hashimoto. U-face: A user interface design system based on multiview model. In *Proceedings of the Fourth International Conference on Human-Computer Interaction*, volume 1 of *Congress II: Design and Implementation of Interactive Systems: INTERFACE DESIGN; User Interface Design Methodology*, pages 684–688, 1991.

[12] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, New York, N.Y., 1994.

[13] the objecttime. Technical report, ObjectTime Limited., 340 March Road Kanata, Ontario.

[14] Rational rose. Technical report, Rational Softwares Corporation.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[16] Technical report, Rational Software Corporation.