CHAPTER 4: HYBRID CONVERGENCE ANALYSIS MODEL

4.1 Introduction

The hybrid convergence analysis model (HYCONAN) was developed taking into consideration two important aspects of systems development, namely the reengineering of legacy systems that was developed using traditional structured methodology and the development of new systems that emphasizes refinement and validation using object-oriented technology and CASE tools. In the hybrid analysis model, the basic requirement is simplicity, refinement and validation at a very early stage of systems development to ensure the smooth transition from analysis to design and to development, and also the development of systems, processes and classes for reuse. The model has been devised to reduce cohesion and coupling among systems, processes and classes. The analysis model emphasizes a two-stage analysis process, called Level 1 analysis and Level 2 analysis. The core concepts in Level 1 are System, Process and Class with Class-Relationship Diagrams (CRD), whereas in Level 2 they are System-Interaction Diagram (SID), Object-Interaction Diagram (OID) and State-Evolution Diagram (SED).

In building the analysis model, five basic principles are applied:

1. The information domain is modeled.
2. Module function is described
3. Model behavior is represented.
4. The models are partitioned to expose greater detail.
5. Early models represent the essence of the problem while later models provide implementation details.
Programming language independence is a means to provide portability of design deliverables across languages, hence giving a greater flexibility to implementers. HYCONAN ascertains this principle by the fact that it embodies most of the underlying concepts of Object-Oriented Programming Languages and Structured Methodology, in order to make a seamless mapping of the design deliverables to the target programming language.

The HYCONAN model does not make a clear distinction between analysis and design tasks. Instead, a continuous process that begins with the assessment of customer specification and ends with design is proposed.

This chapter is aimed to provide a description of the underlying concepts of HYCONAN’s diagramming techniques. An effort is made to provide as many illustrations as possible.

**LEVEL 1 ANALYSIS**

In Level 1 analysis, only the system that is to be developed is analyzed taking into account the interactions of the system with other systems, if there are any. Level 1 analysis emphasizes simplicity to ensure zero coupling and cohesion among components of systems for future reuse. The goal of Level 1 is to find or create system, processes and classes that are broadly applicable, so that they may be reused. Level 1 Analysis is mainly performed when we wish to reengineer legacy systems that were developed using structured methodology to object-oriented methodology.
In Level 1 analysis, we use the following paradigms:

a) System Analysis
b) Process Analysis
c) Class Analysis using the CRD model described below. The intent here is to define all classes (and the relationships and behavior associated with them) that are relevant to the problem to be solved. To accomplish this, a number of tasks must occur:

1. Basic user requirements must be communicated between the customer and the software engineer.
2. Classes must be identified (i.e., attributes and operations are defined).
3. A class hierarchy must be specified.
4. Object-to-object relationships (object connections) should be represented.
5. Object behavior must be modeled.
6. Task 1 through 5 are reapplied iteratively until the model is complete.
4.2 Level 1 Analysis Concepts

4.2.1 System

*Notation* – Rectangle

![Rectangle](image)

*Semantics* – End-user functionality provided by one or more programs consisting of a collection of inter-operating processes.

- A system can be made up of one or more processes (Composition-see *Figure 4.1*).
- A system can interact only with other system. Only one system can exist in any application under consideration.
- A system must have at least one process.
- Each system encapsulates a system name and one or more processes.

Composition is represented by a solid line.

![Composition Diagram](image)

*Figure 4.1: System with its component processes*
4.2.2 Process

Notation – Polygon

Semantics – A process is a major activity that cannot exist on its own. It should be part of a system. It may not communicate or interact with other systems. Processes perform a task that meets the goal requirements of a system. A process may exist in one or more system, that is, the process could be reused in other systems, and it must be composed of one or more classes. If a process performs more than one task then it is broken down into two or more processes to ensure that the processes are simple. A task is an minor activity that is granular and that which cannot be decomposed further. A process may consist of a number of tasks. For example in a payroll system we could have the following processes:

- Input New Employee
- Calculate Salary
- Print Checks
- Print Payslips

Each one is a process within the payroll system that accomplishes a significant goal requirement of the payroll system.

Each process encapsulates a process name, one or more classes and any constraints (any limitations that are imposed on a process).

Composition is represented by a solid line.
Example

![Diagram showing Process A and Process B with Class 1, Class 2, and Class 3 connected to Composition]

*Figure 4.2: Process with its constituent class(s)*

In *Figure 4.2* each of the processes should be analyzed separately in the process-class diagram.

### 4.2.3 Class (CRD)

*Notation* – Round Rectangle

*Semantics* – classes can only interact with one or more classes.

- The smallest unit of a process that performs the goals of a process i.e. the atomic unit of a process.
- A class can be inherited from another class of the same process or a class of another process.
A class can only interact with classes of other processes within the same system. A class is represented by a round rectangle that has four compartments.

Figure 4.3 shows the graphical representation of a class, where the elements in square brackets are optional. The specification of {Abstract} signifies that the class is abstract. The specification of << Parameters>> signifies that the class is generic, whose list of parameters are separated by a comma and are specified within <<>>. The specification of <Arguments> signifies that the class is instantiated from the generic class. Its list of arguments to be passed to the generic class are separated by a comma and are specified within <>.

The symbol "||" denotes that a class can only either be an abstract class, a generic class or an instantiated class, if it is not a concrete class. The three dots (i.e., "...") denotes that the class may have many attributes and operations.

![Class name](image)

**Class name** [{Abstract}] [[<<Parameters>>] [<<Arguments>>]]

**Characteristics**

[Type] Attribute name [= Initial value] [[{External visibility}][{Constraints}]] [.].

[Return type] Operation name[type 1 Par 1,. Type n Param n] [Operation type] [[{External visibility}][{Constraints}]] [.].

*Figure 4.3: Graphical Representation of A Class in a CRD*
The first compartment contains the name of the class (written in bold) and optionally its type which can either be **abstract** or **generic** (concrete by default). An abstract class is a class that cannot have direct instances but whose descendents (subclasses) can have instances, whereas a concrete class is a class that may have instances. A generic class is used as a template to produce other classes, which will thus be its instances when its parameters are assigned values. A generic class may not have any instances and may not itself be used as a parameter.

The second compartment contains a set of characteristics:

**Tangibility.** Does the class represent a tangible thing (e.g., a keyboard or sensor), or does it represent more abstract information (e.g., a predicted outcome)?

**Inclusiveness.** Is the class **atomic** (i.e., it does not include other classes) or is it **aggregate** (it includes at least one nested object)?

**Sequentiality.** Is the class **concurrent** (i.e., it has its own thread of control) or **sequential** (it is controlled by outside resources)?

**Persistence.** Is the class **transient** (i.e., it is created and removed during program operation); **temporary** (it is created during program operation and removed once the program terminated); or **permanent** (it is stored in a database)?

**Integrity.** Is the class **corruptible** (i.e., it does not protect its resources from outside influence) or is it **guarded** (the class enforces controls on access to its resources)?
By default a class is permanent, atomic, tangible, concurrent and guarded. These characteristics would become useful for future class modifications or enhancements. For example, if a class is currently transient, and in future the systems engineer would want to convert it to a permanent class in order to store the class information in a database, then the necessary arrangements have to be made. If a class is currently sequential i.e., it can only be activated by an external trigger, then, in order to convert it to a concurrent class, the necessary attributes and operations modifications has to be done.

The third compartment contains attributes declarations, each of which may include: a type, the name of the attributes, an initial value, external visibility and constraints. A type can be a standard type (i.e., integer, character, real, string, boolean) or any user-defined type. The name of an attribute must be unambiguous in the context of the class and must be unique within the class. External visibility is mentioned to show how a property (i.e., an attribute or an operation) of an object is visible to its clients (i.e., those seeking services). It has mainly two forms: exportable, where the property is accessible to all the clients of the class (which is the default case), and non-exportable, where the property is accessible only to the class itself. A constraint is an expression of some semantic condition that must be preserved.

The fourth compartment contains operations declarations, each of which may include: a return type followed by the symbol "<->" which is used to separate between the name of the operation and the return type, the name of the operation, a list of parameters and their types, the operation type, external visibility and constraints. An empty parameter list within parenthesis shows explicitly that the operation has no-parameters, or that no decision has yet been made about the parameters.
Each operation has an operation type. A **static operation** operates on the class as a whole rather than on any single instance of the class. An **instance operation** is shared by each individual instance of the class. A **constructor** operation creates instances (i.e., objects) of a class. A **destructor** operation deletes instances (i.e., objects) of a class. A modifier operation may change the values of an object's attributes, and thus the state of an object. A **selector** operation does not change the values of an object's attributes, and thus does not change the state of an object. An **abstract operation** is an operation that is declared but not implemented by an abstract class, it is rather implemented by its subclasses. If the operation type is not specified for an operation, no decision has yet been made for that operation.

*Figure 4.4* below shows an example of a class named Person, where no decision has yet been made about the possibility of being an abstract or a generic class, since the class stands alone at the early activity of identifying classes in the analysis phase. The class has six attributes: *Name, Age, Identification number, Job Title, Address and Salary*. The class has two operations: *Change job* and *Change address*.

![Person Class Diagram]

*Figure 4.4: A class instance*
Figure 4.5 below shows an example of a generic class identified in the design phase with its instantiated classes. An instantiated class is connected to its generic class by a dashed arrow. In this example, the attributes of the classes are hidden. From the generic class Vector, with two parameters, Type and Size, two classes are instantiated, each one deals with a particular type of Vector (i.e., Integer and Real) and a certain size (i.e., 1 and 2) of Vector.

![Diagram showing generic class and its instances](image)

*Figure 4.5: A generic class and its instances*

4.3 Relationships

4.3.1 Class Inheritance (Generic-Specific)

This is a relationship where one class is an extension of another class. Inheritance allows a class to take on, or inherit, all the features of another class. The class being inherited from is known as the generic class, while the class that inherits is known as the specific class (see Figure 4.6). The specific class inherits the attributes and operations encapsulated by the generic class, meaning that the specific class includes the generic attributes and operations as part of itself. In addition, the specific class can add new attributes and operations. Also, the specific class can redefine or override the implementation of an inherited operation so as to be able to specialize it to the needs of the specific class. The specific class is an extension or specialization of the generic class.
We can talk about inheritance as the is-a relationship, as the specific class is everything the generic class is, plus some new bits. For example, if we have a class *Vehicle*, a specific *Car* can be defined as an extension of *Vehicle*. We can then say that a *Car* is-a *Vehicle*; a *Car* is everything a *Vehicle* is along with a set of additional features.

Generic-Specific relationship is represented by a directed solid-line.

Specific classes would appear with solid borders as shown below.

![Diagram](image)

*Figure 4.6: Generic and specific classes*
Example

E.g. Cash and Credit customers might be specific classes/objects of the generic class customer.

![Diagram of generic customer with specific Cash and Credit Customer](image)

*Figure 4.7: A generic customer with specific Cash and Credit Customer*

4.3.2 Possession

Possession relationship is depicted by having two arrow heads beside the composite class. Possession may be *logical* or *physical*. Physical possession occurs when a class is a physical part of another class (e.g., a car and its wheels), whereas logical possession occurs when a class is a logical part of another class (e.g., a document and its paragraph). Physical possession may be by *reference* or by *value*.

Physical possession by value means that the component class is an attribute of the composite class, whereas physical possession by reference means that the composite class contains a reference (i.e., a pointer) to the component class. Physical possession by value implies that the construction and destruction of the component objects occurs as a sequence of the construction and destruction of the composite object. Thus, the composite and its components have the same life cycle.
Logical possession and physical possession by reference imply that the life cycle of the composite object and its component objects are independent. Logical possession and physical possession by reference are implemented in the same manner (i.e., using pointers), but at the analysis level, these are two semantically different concepts. By default, a possession relationship is logical.

A physical possession by value is represented by a \{V\} beside the component class, whereas a physical possession by reference is represented by a \{R\} beside the component class. Cardinality may also be shown beside the component class.

The Figure 4.8 shows an example of a logical possession, where a composite Document contains many Paragraphs, containing in turn many Sentences.

![Diagram](image)

**Figure 4.8: Logical Possession**
The Figure 4.9 shows a recursive possession which is a combination of a possession and an inheritance. A computer program logically contains many blocks. A block can either be a compound statement or a simple statement. A compound statement in turn contains logically many blocks.

![Diagram of recursive logical possession]

Figure 4.9: Recursive Logical Possession

The figures 4.10(a), 4.10(b), and 4.10(c) show some examples of physical possession. A polygon physically contains by value a graphic bundle and physically contains by reference three or more points. A company physically contains by value many divisions, in turn a division physically contains by value many departments.

![Diagram of polygon with its possessions]

Figure 4.10a: Polygon with its possessions.
For example, a toaster consists of 3 parts such as the coil, elevator and the chassis. Every toaster must have 1 coil, 1 elevator and 1 chassis. The relationship is depicted diagrammatically below.
4.3.3 Association

An association is a non-hierarchical relationship between classes representing a semantic dependency. A link is an instance of an association, and thus a relationship between instances (i.e., objects) of the related classes. Objects communicate via messages through links. Association is not a new concept since it has been widely used throughout the database community for years, however, it is a useful construct for all kinds of system.

Associations are inherently bi-directional. The name of a binary association (i.e., association between two classes) usually reads in a particular direction, but it can be traversed in either direction. A binary association is represented by a double arrow joining the two related classes as shown in the figure below. The name of the association is written on the double arrow along with the symbol, "->" or "<-", which indicates the reading direction of the name of the association.

The role is one part of an association, whereas a role name is a name that uniquely identifies one end of the association. Roles provide a way of traversing a binary association from a class to another, without explicitly mentioning the name of the association. It is often easier and less confusing to assign role names instead of (or in addition to) the name of an association. Role names must be mentioned in an association within the same class (i.e., recursive association). If there are many associations between the same two classes, role names are written beside the corresponding association and down to the double arrow. Associations are often left unnamed when they can be easily identified by their classes. Roles names must be unique and should not have the same name as an attribute of the related class. Cardinality specifies the number of instances of a class that may participate in a relationship and can take the values as follows:
i exactly 1
+i i or more
i1..i2, i3 A specific range or a value

Note: i, i1, i2, i3 are any integers
By default the cardinality is many (+() )

The Figure 4.11 shows an example of two associations. In the association Employs between the classes Company and person, the company plays the role of Employer and the person plays the role of Employee. The association is many to many, which means that the company employs many persons and a person can work in many companies.

The association Manages is an example of a recursive association, where a person can play either the role of a Worker or a Boss. Cardinalities are mentioned as: a boss can have many workers and a worker can have zero or one boss. In this example, the operations of the classes are hidden.

![Diagram](image-url)

*Figure 4.11: An example of an association relationship*
4.4 **LEVEL 2 ANALYSIS**

The HYCONAN model does not begin with a concern for objects. Rather, it begins with an understanding of the manner in which the system will be used – by people, if the system is human-interactive; by machines, if the system is involved in process control; or by other programs, if the system coordinates and controls applications. Once the scenario of usage has been defined, the modeling of the software begins.

In Level 2 analysis, we consider the behavioral aspects of the system, process and classes. In this level, we consider the interactions of systems with external systems, among processes and among classes within the same system.
A system cannot spontaneously activate itself, nor can a process or a class. Activation of processes or classes only can happen through a system. Activation of systems may be in the form of external inputs such as user input, hardware signals, program triggers via data or service request. These external triggers generate messages. A system will first receive a message from an external source, which it then transmits, to the appropriate process. A system could activate one process for each message at any one time or it could generate a process interaction among processes within the same system. When a process receives a message, it sends it to the required class that performs the necessary tasks and sends the output back to the process and back to the system that first generated the message trigger. The system will then send the output message to the other system that requested the service in the first place (see Fig. 4.11(a) and 4.11(b)).

![System Interaction Diagram]

*Figure 4.11(a): Example A: System B sends a message to system A which then sends the message to process 1. Process 1 then activates a single class to perform the necessary task.*
Figure 4.11(b): Example B: System B sends a message to system A which then sends the message to process 1, which then triggers a class interaction to perform the necessary task. Class 1 and Class 2 may or may not belong to the same process of a system.

In Level 2 analysis, we use the following paradigms:

a) System Interaction Diagram (SID).
b) Object Interaction Diagram (OID).
c) State Evolution Diagram (SED).

Steps to perform when using the hybrid analysis model at Level 2:

Step 1: Analyze each system and identify all interactions. Refine and validate with users if necessary. A system can only activate one process for each message. Draw the state-interaction diagram (SID).
Step 2: For each process, identify the class or class(s) that is (are) activated. A process may activate one class that may interact with one or more classes to produce the required output. If there are any interactions among classes, identify the interactions and draw the object-interaction diagram (OID).

Step 3: For each state change of an object of the class, draw the state-evolution diagram (SED).

4.5 LEVEL 2 ANALYSIS CONCEPTS

4.5.1 System Interaction Diagram (SID)

![System Interaction Diagram](image)

Figure 4.12: System interaction diagram

The system under consideration is called the focus system. The SID is used to show at a higher level all interactions of the focus system with external systems. It represents a snapshot in time of a stream of messages over certain configuration of interacting systems. It is also used to trace the execution of a scenario, which is a sequence of messages passed between systems during one particular execution of a message.

In Figure 4.12, System B sends a message to system A (focus system) which passes the message to process 1. If a message is being sent to another system, then that system needs to have one or more processes to carry out the appropriate processing.
A system could generate many messages to many processes within the same system, but only one message could be passed to a single process at any one time. An external system could also request or generate more than one message to more than one process as shown in Figure 4.13 below. The processes of the focus system are visible to external systems and vice-versa.

![Diagram showing system interactions](image)

*Figure 4.13: System B triggers two processes in System A.*

A system could take on the role of a client or server system. A message flow between two systems could invoke a process in another system. The sequence of message flow could also be represented with an indication of whether the system is triggering the message or responding to a message. A message flow by default is a trigger and if no sequence number is indicated then the message flow is independent of all other message flow and is time independent or *asynchronous* (i.e., it could be triggered at any time).
Figure 4.14: Diagramming elements of a system interaction diagram (SID)

Message = (Operation sequence number)(Status)
Process name (Parameters)
Status = \{T\} (Trigger), \{R\} (Response) or \{A\} (Asynchronous)
i = any integer value > 0. (Operation sequence number).
Parameters = \{return type\}<return value>

Example

Branch Campus
Student
Registration
System

\{1\}\{T\} student info (string id)

student details \{2\}\{R\}

\{A\} Update database (File student file)

\{A\} Fee Processing (File Student file)

Student Info
Update Database

Student Management System
(Focus System)

Accounting System

Figure 4.15: System interaction diagram for the Student Management System
Figure 4.15 above represents a simple university student record system. The focus system is the Student Management System. The Accounting System and the Branch Campus Student Registration System are external systems. The focus system is composed of two processes, namely update database () and the student info (). The branch campus Student registration system first requests for a particular student information by sending the student id, the Student Management system then responds by sending the appropriate student details. The activities are sequential. The branch campus student registration system also could send student files for update at anytime to the Student Management System. The student Management System could also send student files to the Fee Processing process of the Accounting System. Only the processes of the focus system is to be displayed in the SID.

### 4.5.1.1 Object-Interaction Diagram (OID)

An OID is used to show the existence of objects and the interactions between them during one particular execution of a system (i.e., for one particular message flow in a process). It represents a snapshot in time of a stream of messages over certain configuration of interacting objects from a process. It is also used to trace the execution of a scenario, which is a sequence of messages passed between objects during one particular execution of a system. OIDs are similar to the protocol diagrams frequently found in communication protocols (Castillo, et al).

OID is a logical continuation of the analysis process in the HYCONAN model. In the SID all message flow to and from the focus system is triggered or responded to by processes within the focus system. All messages from a process must go to at most one class first. Each message trigger may trigger a state change in the object of the class.
A process can only send a message to a single class within the same system under consideration. Each message trigger would invoke a particular operation in a class. A process can send many messages to many classes within the same process but only one message to a single class at any one time.

When a class receives a message from a process, it may or may not interact with other classes within the system. Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility. A collaboration implies either a request for information or a request for some action. The output of this interaction may or may not be given to a process, depending on whether the process that generated the message requires a response.

In an object interaction, two objects are involved: a client, the object invoking an operation (class or process) and the server, the object supplying the operation. The sender of a message knows the receiver, but the receiver does not necessarily know the sender. Generally, a message passing represents an operation invocation. In a OID, objects are represented by round rectangles, processes by ellipse and message passing by arrows from the client object to the server object. The name of the object's class is written inside the object round rectangle as shown in the figure below.

![Diagramming Elements of an OID](image)

**Figure 4.16: Diagramming Elements of an OID**

Act = Active object
Class Message = [Return type \(\rightarrow\) Operation name [(Arguments)]][(Operation sequence number)]{(Synchronization)}{(Visibility)}{(Constraints)}
Synchronization = \{S\} (Synchronous), or \{B\} (Balking), or \{T\} (Time-out), or \{A\} (Asynchronous)
Process Message = [Return type][Return argument][(Operation sequence number)]
Objects may have a purely sequential semantics, which is guaranteed only in the presence of a single thread of control (i.e., one active object at the same time). In this case, the ordering of messages passes between objects is important. Objects may also be active, meaning that they embody their own thread of control but their semantics is guaranteed even in the presence of multiple threads of control, by synchronizing the message passing using the following communication protocols (Booch, G, 1991):

- **Synchronous** (represented by a \(\{S\}\)) : An operation commences only when the client object has initiated the message and the server object is ready to accept it. The client and server objects wait for each other until they are both ready.

- **Balking** (represented by a \(\{B\}\)) : Similar to synchronous, but the client object will abandon the operation if the server object is not ready.

- **Time-out** (represented by a \(\{T\}\)) : Similar to synchronous, but the client object will only wait for a specific amount of time for the server object to be ready.

- **Asynchronous** (represented by a \(\{A\}\)) : An operation commences regardless of whether the server object is expecting the message or not.

In an OID, an object is sequential by default, unless otherwise stated active using the keyword \(\{Act\}\). The presence of active objects imply concurrency, where messages are passed at any time and at any order. Operations sequence numbers are included in the case of sequential scenarios, whereas synchronization aspects are included in the case of concurrent scenarios.
A scenario of a warehouse loading system and its corresponding sequential OID is shown below.

1. The goods despatcher informs the clerk that the goods have arrived.
2. The clerk tells the system that the loading bay is empty.
3. The clerk gives the system details from the manifest.
4. The clerk starts checking in the goods.
5. The system issues an identifier for each goods package.
6. The system informs the clerk of any discrepancy in the delivery, according to the manifest.
7. The system assigns the store building where the goods are to be stored and sends the allocation to the warehouse supervisor.
8. The system tells the clerk, which goods cannot be stored and must be returned from where they come i.e., to the despatcher.
9. The goods are returned to the despatcher.

Figure 4.17: The Sequential OID of A Warehouse Loading System
A concurrent OID shown in figure 4.18 below is related to the automation of a electric fryer. All the objects in the diagram are active except the object Light which is sequential. The *Startup* operation is synchronous, the Electric Fryer has to wait for the Heating Coil to be ready to perform the operation. The *Failure* operation is asynchronous, the Heating Coil does not wait for the Electric Fryer to be ready to perform the operation. The *Start* operation is time-out, the Electric Fryer has to wait for a certain amount of time for the Thermostat to be ready to perform the operation. The *Is ready* operation is balking, the Thermostat does not wait if the Electric Fryer is not ready to perform the operation. Finally, the operation *Turn on* is sequential since it is performed by the sequential object Light.

![Diagram](image)

*Figure 4.18: Concurrent object interaction diagram*
4.5.1.2 State-Evolution (SED)

*Notation* – Circle

A state evolution diagram (SED) is a standard computer science concept (i.e., a graphical representation of finite state machines) that has been handled in different ways in the literature, depending on its use.

A SED is used to show the state space of a given class, the events that cause a transition of object from one state to another and the actions that result from a state change. A SED is a graph whose nodes are states and whose directed arcs are transitions labeled by events and actions.

4.5.1.3 Definitions

A state of an object is an abstraction of its attribute values at a certain point in time. An individual object’s state can be generalized to apply it to its class, because all instances of the same class live in the same state space, which encompasses a finite number of states (Booch, G, 1991).

A state is depicted by a circle containing the name of the state in bold, an indication in braces of whether it is a start state, a stop state or a deadlocked state (i.e., a transition in which no further state change can occur), the event that causes the entrance to the state if it is a start state, activities preceded by the keyword Ac: and recursive transitions (i.e., transitions from a state to itself) (see figure 4.19). A recursive transition can either end-up in an iterative state or a temporary state until some event causes a state transition.
An activity is an operation that takes time to complete. It is generally associated with a state. Activities include continuous operations like ringing a telephone bell, as well as sequential operations that terminate by itself after an interval of time, like a robot moving a part. Activities correspond to operations in a class.

A transition is a state change caused by the occurrence of an event. A transition is depicted by an arrow from one state to another. The name of the event, any condition on the event, the symbol "\rightarrow" separating the event and the action, the resulting action with its arguments and its corresponding operation are labeled on the arrow as shown below.

![Diagram](image)

**Figure 4.19: Diagramming Elements of a SED**

An event is a problem domain stimulus that occurs at a certain point in time. Events represent pre-conditions in the specification of an operation or part of an operation in an OOPL.

An action is an instantaneous operation associated with an event. An action may be an operation on its own or part of an operation. If the action is part of an operation, the transition specification is enhanced to show the operation to which this action belongs.
If an event causes a transition from a state before an activity in that state is completed, then the activity is terminated prematurely. If an object is in a state and an event labeling one of its outgoing transition occurs (i.e., the first one to occur), the object enters the state on the target end of the transition which is said to be fired.

Given a state S, with transition T upon event E with condition C and action A, the following order applies: event E occurs; the condition C is evaluated; if C evaluates true, then T is fired and action A is performed. The lack of event label indicates that the transition is fired automatically when the activity in the state is completed.

The Figure 4.20 shows a SED of a vending machine. The occurrence of the event Coins in triggers the action set balance(Amount) which changes the state of the vending machine from Idle to Collecting Money. In the Collecting Money state, the occurrence of the event Coins in triggers the action Add to balance(Amount) without a state change (i.e., a recursive transition), whereas the occurrence of the event Cancel triggers the action Refund coins that changes the state to Idle.

The occurrence of the event select item triggers a transition to the unnamed state containing the activity Test item and compute change.

Being in the unnamed state containing the activity Test item and compute change, if the condition Item empty or change < 0 is evaluated to true, a transition to the state Collecting Money is fired, whereas if the condition Change=0 or Change>0 is evaluated to true, a transition to the unnamed state containing the activity Dispense item or to the state containing the activity Make change is respectively fired.

When the activity Make change is completed, a transition to the state containing the activity Dispense item is automatically fired. Similarly, when the activity Dispense item is completed, a transition to the state Idle is automatically fired.
Figure 4.20: A SED for a Vending Machine

4.6 Conclusion

Although the ideal process for adopting object technology into commercial information systems development environments is to replace existing functional decomposition and procedural implementation tools and methodologies with their object-oriented equivalents (an O-O-O approach), thus providing a "seamless" transition throughout the lifecycle, it is recognized that many software developers will take a more pragmatic approach. In this they will incrementally adopt the emerging technology by deciding to "mix and match" object-oriented techniques with functional decomposition techniques to create a hybrid object-oriented/functional decomposition software systems development methodology. This pragmatism may be a result of the large current investment in top-down functional decomposition, in terms of both expertise and front-end CASE tools. Secondly, even if the OO design methods are viewed as "more-natural", the investment in code, say, COBOL leads to the question, Can an object-oriented design (viewed as a better design) be

113
implemented in a non-OO language? The two most likely paths are to adopt object-oriented design and implementation while retaining functional specification and analysis (Booch, 1987) (F-O-O methodology - Henderson Sellers, 1991), or to retain implementation in a procedural language (for code compatibility) and precede this with object-oriented analysis and design (the O-O-F methodology). It is understandable to one who has been imbibed in either one of the two methodologies (OO and F-O-O), not to appreciate the mix-and-match approach.

The rationale for the F-O-O methodology is that since much of the enthusiasm in object techniques was generated in the latter stages of the life-cycle with the advent of new object-oriented programming languages (OOPLs), software developers may have adopted the new paradigm at implementation and probably design stages. The hybrid model encompasses both the structured and OO paradigms and hence concepts such as systems and processes for structured paradigm and classes and states for the OO paradigm has to be retained. If systems and processes were to be removed from the hybrid model then the model would be purely OO in nature.

(1). If an organization has developed systems using structured methodology and now would like to use OO techniques and OOPL for implementation (F-O-O).

(2). If an organization would like to do OO analysis and then implement using a procedural language (O-O-F). The classes could be reused in case implementation is to be done using OOPL.

For example, let's say an organization would like to use COBOL and take the F-O-O path. They would have to draw the SID, CRD, OID and SED. In this case the classes would be used as a subprocess (or modules) in the implementation and the process would be the subprogram and program chaining could be applied to form a system. If in case at a later stage the organization would like to use OOPL, there is no need for further analysis and the classes could be reused. The analysis model is bi-directional for implementation.