Lecture Notes in Computer Science: Metrics for Test Plan on Object-Oriented Behavioral Design

Young Chul Robert Kim 1 * and C. Robert Carlson 2

1 Embedded System Lab
LG Industrial System R&D Center
robertkim@lgis.com

2 Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
IA_CARLSON@alph1.iit.edu

Abstract This paper describes a design driven approach to test plan generation. Its foundation is an object-oriented software design approach which partitions design schema into a layered architecture of functional components called "design units". A use case software development methodology is employed which preserves this unit architecture on through to the actual code structure. Based on the partition design schema produced during the design phase of this methodology, a test plan is generated which includes a set of unit and scenario based tests. A software metric is introduced which produces an ordering of this set to enhance productivity and both promote and capitalize on test case reusability. This paper contains an application which illustrates the proposed approach.

1.Introduction

This proposed paper is based on the use case approaches defined by Jacobson [6], UML, Carlson [3] , and Hurlbut [5]. Specially, Iva Jacobson told me in Seoul's lecture that most persons focus on reusability of source code level, but should consider reusability on all software development activities. Therefore, focusing on design phase, this idea begins by considering possible ways to enhance Hurlbut's action matrix concept [5] bring software testing concepts into design. Hurlburt's notion of an action unit needs to be refined based on a conceptual analysis of the method sequences found in sequence diagrams (called message interaction diagrams), which, by the way, is the first significant artifact produced during use case design. Our preliminary analysis of this issue has resulted in the introduction of the concept of a "design component". Several definitions for a design component have emerged from our research for the designer to choose from depending on the level of abstraction desired and the preference for testing techniques to be applied. Even some definitions of this component guilds to generate skeleton code. To provide an automated process by which an action matrix can be produced, thus, leaving the designer with only an ad hoc, manual approach, we need to develop an algorithm to produce the action matrix from sequential diagrams (called Interaction diagrams). Such an algorithm would significantly improve the productivity of the designer in producing such potentially useful information. This conversion algorithm is developed to identify or extract each type of design unit from the basic interaction diagram in which deals with single control object and the other containing multiple control objects.

The action unit component of an action matrix is defined as specific design units of sequential diagram (called interaction diagram) for easily identifying reusable design component with test plan metrics. Several possible definitions of design units is introduced, each processing different testing characteristics. The scenario component of a action matrix is defined for the purpose of generating a preliminary test plan.

2. A layered design unit architecture

This idea is based on object-oriented behavioral design which partitions design schema into a layered functional components called "design unit". Based on the partition design schema produced during the design phase of use case methodology, we can find the information of reusable design units, and generate test plan which includes a set of unit and scenario based tests on early stage rather than test stage.

2.1. Definition of Design units

One of the first artifacts produced during the design unit phase is a sequential diagram based on use case methodology. The message sequence defined by the sequential diagram can be partitioned into a sequence of design units. As a result, different testing techniques may be appropriate depending on the choice of deign unit. Based on a this layered architecture, we can also identify the design reusable components through object-oriented behavioral design of use case methodology. Therefore, this provides the design and test team member with different options to choose from at design time. Several partitioning strategies are possible, each resulting in a design unit specification that exhibits different characteristics, particularly at boundary points between adjacent design units.

<u>Method unit</u>: method executed by an object in response to a message

Consider the simple sequential diagram described in Figure 1. In this case there are six methods labeled m1,m2,m3,m4,m5, and m6. There are the distinctive design units.

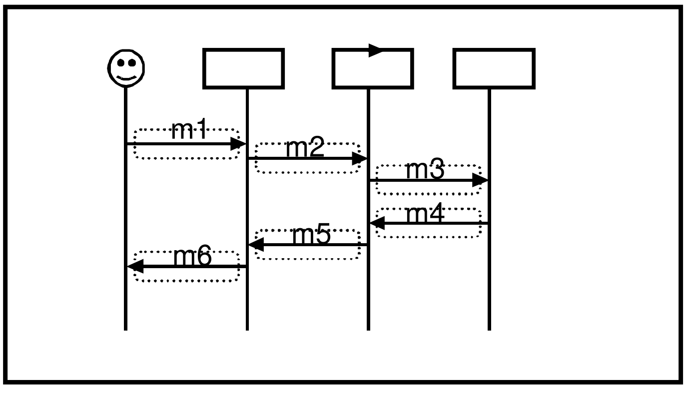
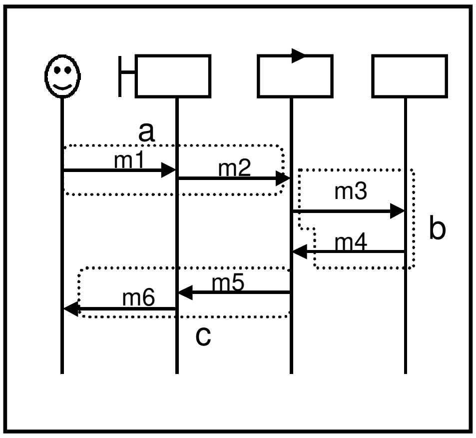


Figure 1. Example of a Method Unit

The simplest choice is to let each method be a design unit. The people assigned to develop this unit, therefore, be in the best position to perform unit testing because of their familiarity with the unit. A drawback to this approach is the large number of design units to be tested. No additional properties of such a design unit can be assumed on which to base a specific testing approach. In this unit case, we may not mention the reusable unit of all design units.

<u>Reusable Pattern Unit</u>: Sequence of methods executed by a particular object pattern. Illustrate a possible grouping of methods based on (presumed) reusable pattern units. This simple vending machine example contains just reusable patterns.



<u>Vending machine</u>:
- method:
  m1: insert coin
  m2: check coin
  m3: measure the weight of coin
  m4: judge what coin
  m5: notify
  m6: notify
- reusable pattern unit:
  a = m1+ m2;
  b = m3 + m4;
  c = m5 + m6;

Figure 2. Example of a Reusable Pattern Unit

If a library of reusable design components was used to develop the sequential diagram (called interaction diagram), then they could be used to define the design units. The major advantage of this choice is that both code and test specifications may already exist for these units. Further, standard interfaces between these design units may already exit, a characteristic that is essential for system evolution. Each reusable pattern unit in this example has a single input and single output (SISO) characteristics. In general, a pattern can have any one of the input and output characteristics described in Figure 3. Good reusable patterns are the ones whose interface supports some degree of standardization. This could have worthwhile implication for the development of standard unit testing practices

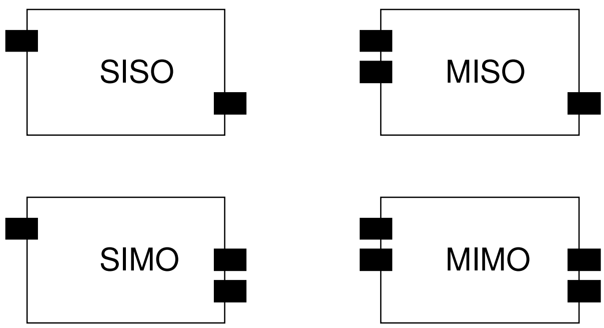


Figure 3. Characteristic interface of SISO, MISO, SIMO, MIMO

The cardinality of pattern's interface is a key factor towards achieving this goal. The cardinality can be applied to the number of message or objects that define the interface. While a pattern may need to support multiple messages, it does not mean that multiple objects are required to define the interface. Greater control can be achieved, particular across multiple system releases, if the interface to and from a pattern is limited to a single object. Of the patterns described in Figure 3, the SISO pattern is preferred because it simplifies the testing process. However, that may not always be possible. What we don't want it for there to be a cross-product of input/output, i.e., all possible combinations M*N in a MIMO pattern. If, however, each input & output are uniquely paired {1-1}, then MIMO can be reduced to a set of SISO protocols as follows: {1-1} {2-2} {3-3} in MIMO in Figure 3.

State Unit: sequence of methods executed during the interval bounded by consecutive states as defined by the corresponding event state model.

illustrates a possible event state transition scenario for a simple ATM machine. Byun's algorithm [2] analyzes the interaction diagram and creates an event state model for subsequent code generation. Our proposed action matrix algorithm follows the same partitioning scheme to provide design units, thus providing the developer with needed information to "standardize" code development practices. We can identify state units as the result of the messages (m) received by control objects and sent from control objects. We can identify each state for a particular object, that is, the control object(s) such as 'S1' and 'S2'. Stable states are important [Carl99] for establishing database consistency throughout a process, a framework for process evolution during subsequent software releases and standardization of system behavior associated with each state.

Simple ATM Machine:

m1: push withdraw button
m2: request withdraw
m3: request a user's status to the main bank computer
m4: check the user's status
m5: withdraw money
m6: money come out

S0: an initial state
S1: a withdraw requesting state
S2: a withdraw state

PC: pre-condition
PS: post-condition

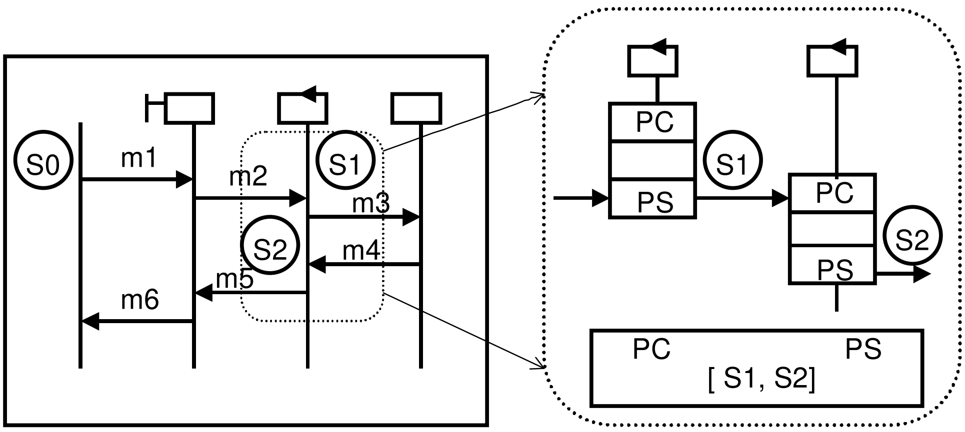


Figure 4. Example of State Units

From a testing point of view, a predicate expression capturing this stability can be associated with each state. A state unit is defined by a sequence of methods executed during the interval bounded by consecutive states. A state unit can be characterized by a state pair (Si, Sj) and can be enhanced by the predicate conditions associated with each of these states. Thus, a testable state unit can be regarded as a triple (Pre-condition **(PC)**, State Pair, Post-Condition **(PS)**). This approach has been coordinated with the parallel research by Byun [2] on the algorithmic generation of event state tables. The implication is that a systematic approach to embedding unit test stubs internal to the event state table driven code is established. Byun's thesis provides a comprehensive analysis of various interaction diagram features and how these features affect the automatic generation of an event state model. The following discussion illustrates how predicated expression associated with an interaction diagram can be used to generate a testable triple associated with state transition pairs.

These representations of the state are assumed to define the behavior of the particular object, that is, the control object(s) in addition to the notation for a choice with possible n-ary branches. In this notation a choice is represented by 'a black colored rectangle bar' on vertical line of the control object.

The proposed state boundary based testing approach will check the condition of a particular object. For example, it checks a precondition when a message comes in. If the condition is satisfied, then it will transfer to the next state. On a decision node with a binary branch, it will transfer to either one state or another state according to each postcondition. Thus, the state boundary based testing will check the precondition of a state pair (Si, Sj). As a result of the condition (that is, PC [S1, S2|S3] PS1|PS2) it will transfer to either S2 or S3 in Figure 5.

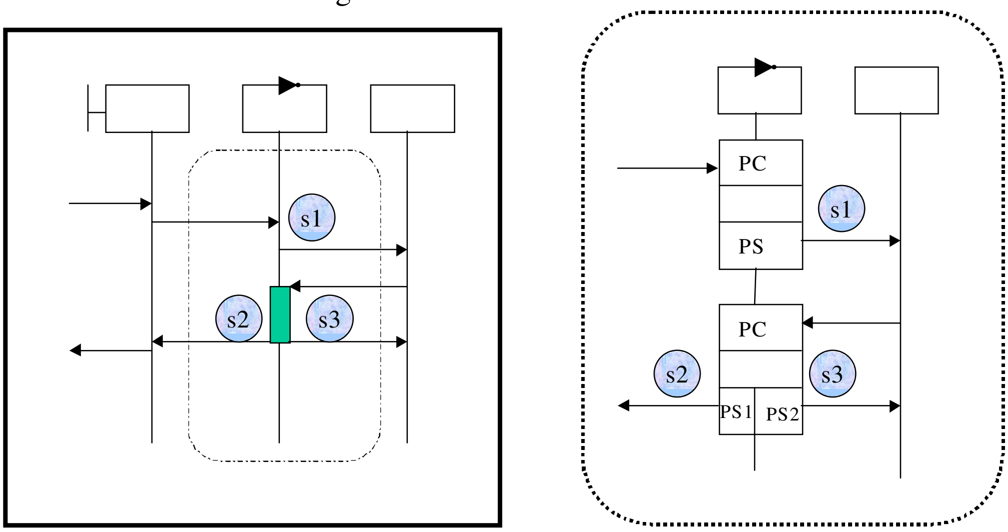


Figure 5. Changing the State on a Decision Node

<u>Maximal Linear Unit (MLU)</u>: a sequence of methods executed during the interval bound by consecutive choices (both actor and object choices).

Consider of this example that an MLU is the same as the dialogue unit if no choice/ branch nodes exist except for the actor.

Figure 6. includes a choice node for the control object. Thus, the MLU for this interaction diagram is (a,b), (c), and (d,e).

The significance of this design unit is that it establishes reference points for straight line testing techniques during design. Without an embedded breakpoint (decision, branch, or collection point) this unit is the same as the dialogue unit. With a decision node within a dialogue boundary point, each MLU is represented by {<Si1, Si2, …Sin>}, that is, a linear sequence of state changes Si1 through Sin and branches before and after.

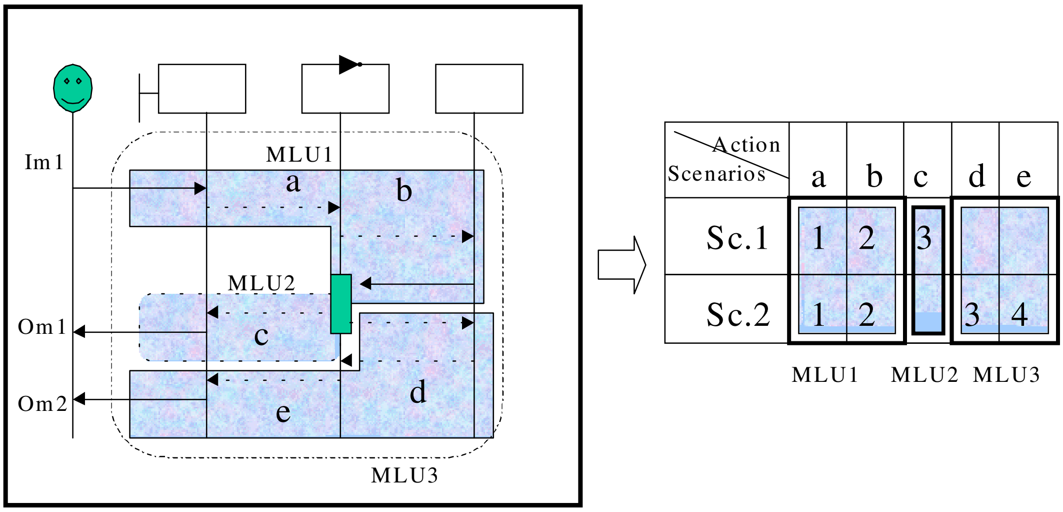
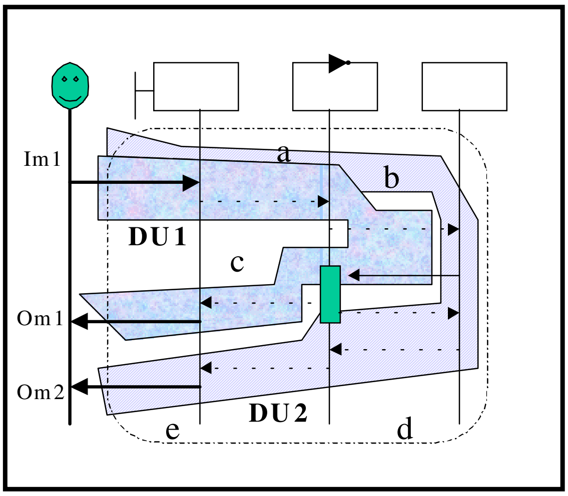


Figure 6. Example of MLUs (Max Linear Units)

<u>Dialog Unit</u>: a sequence of methods executed during the interval bounded by input from an actor and the response to that actor.

In an interaction diagram with the choice notation of Figure 7, sequences of messages / methods are executed during the interval bounded by input from an actor and the response to that actor. We can identify all possible paths from an actor through the system to itself such as dialogue 1 ('a b c'), dialogue 2 ('a b d e'), and so on.

This design unit has significance for establishing user (actor) acceptance test specification during design. These specifications should be a refinement of user acceptance test specifications defined by use case specifications during the requirements phase. We can also consider a reusable design pattern of this deign units. As part of the requirements tracking process, each use case should be tracked to a set of dialogue units at design time. DU represents {< Mi1, ……. Min>}, that is, the actor initiates Mi1(message) and receives Min (message) and all other message are internal. Dialogue is (Im, Om) where Im is an input message and Om is an output message on the actor.



Action / Scenarios	a	b	c	d	e
Sc.1	1	2	3		
Sc.2	1	2		3	4

DU1:(Im1, Om1) = {a, b, c }
DU2:(Im1, Om2) = {a, b, d, e }

Figure 7. Example of Dialogue Units

3. Refined Action Matrix Approach

From this point, we will explain with one case study based on a 'Restaurant application' use case in Figure 8. Focusing on the customer actor's view, there are three high-level use case scenarios such as the reservation, the normal service, and the carry-out process.

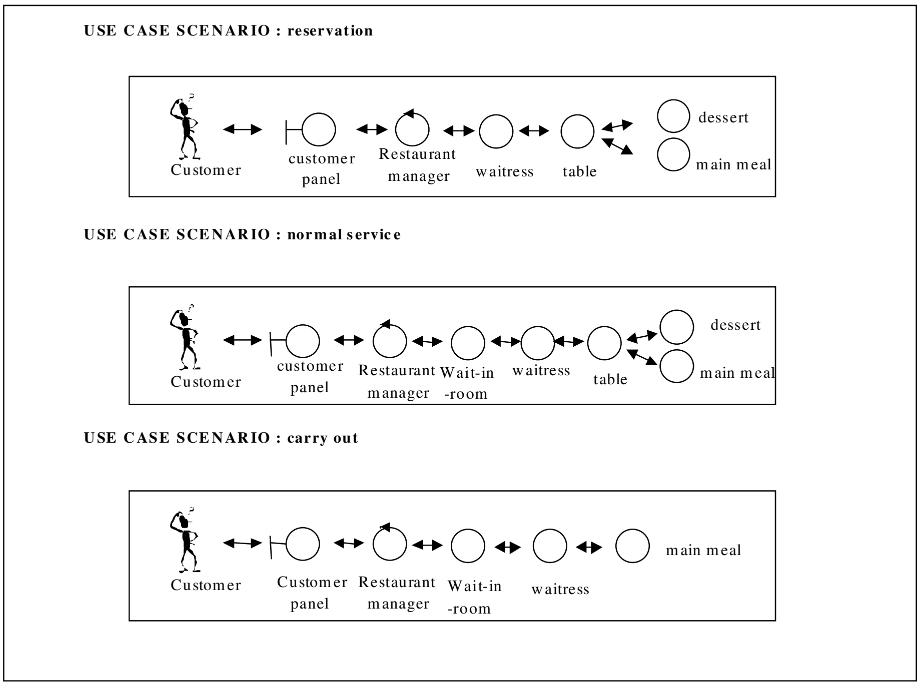


Figure 8. High-level Use Case Scenario 'Restaurant Service' Application

From the requirement specification and the high-level use case scenario analysis, we can design the sequential diagrams (called interaction diagrams) through passing several steps. In this paper, I will skip several steps to develop three sequential diagrams from high-level use case scenarios showed in Kim[11,12] With these sequential diagrams in Figure 9, I can convert the action matrix and use case map dialog through producing these sequential diagrams based on high-level use case scenarios at design stage. One of these sequential diagrams is 'Reservation use case' in Figure 9.

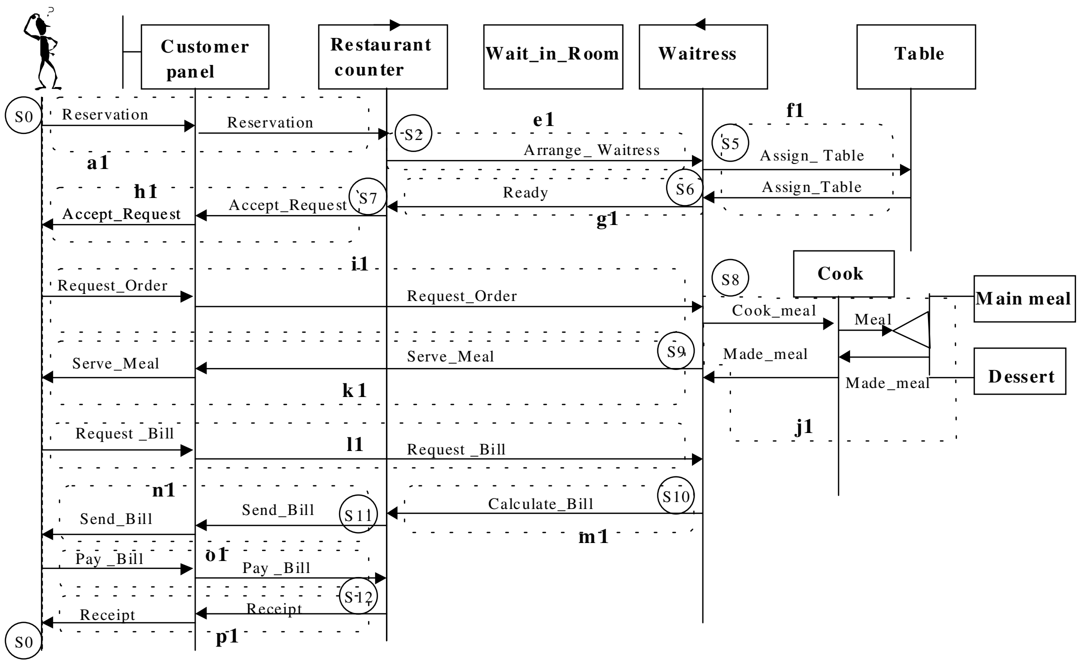


Figure 9. Design units of the Reservation sequential Diagram

What is action matrix? Hurlbut [5] noted that an action matrix presents a cross-match between each action that is included in a use case with each scenario that performs the action. scenario includes an ordered set of actions that explains its course of actions. The use case action matrix is intended to present the scenario designed in a tabular form as the main course of action as a collection of actions that shows the coverage of its use case action by all the variant scenarios in Figure 10 (a). Hurlurt also mentions that an alternative representation of the matrix is a use case dialog map in Figure 10 (b).

Semantics: Each scenario consists of an ordered collection of *action units.* The extended action matrix is intended to tabularly represent the scenario designed as the main course of action unit as a collection of cells that shows the coverage of its use case action unit by all variant scenarios. Each *action unit* consists of the cluster of the consecutive messages (methods, the basic actions) triggered, also relates between the state and next state, and involves the particular objects which send these messages.

Notation: In The first row of a matrix, each cell is matched to a unique use case *action unit* and scenario combination. The rows of a matrix are assigned as scenarios and the columns are assigned as action units. If a use case action unit is also included in the scenario, we make a one-to-one correspondence between cells and integers which results in sequential ordering of the use case action units in the scenario. When a single use case action unit appears more than once in a scenario, we can assign multiple not necessarily contiguous integers in each cell. Since a bunch of the consecutive use case action units may appear iteratively in a scenario, we may parenthesize a bunch of integers that are related with consecutive use case units.

Presentation Option: An alternative representation of the matrix is possible by converting use case action units into nodes in a direct graph with the Mealy's and the Moore's Finite State Machine as well as Musa's Operational Profile concept. When the matrix is presented in this fashion, it may be alternatively refereed to as a use case dialog map that is mentioned by Hurlbut [Hurl98].

Mapping: Each column maps to a use case action unit. Each row maps to a
scenario. In a use case dialog map, we assign each probability of occurrence to each link as a weighted probability value, which is adopted by Musa's Operational profile [9,10]. Musa's approach is a quantitative characterization of a system, which is retained for the most-used part of the system and is reduced for lesser-used parts with the amount of reduction related to the difference in usage. That is, Musa's operational profile is frequently weighted by criticality that reflects both how the system is being used and the relative importance of the uses. We may either guess the probability of occurrence on each branch of the specific node (action) or survey the collection of data [3,9,10,12,13].

Action unit \ Scenario	a1	b1	c1	d1	e1	f1	g1	h1	i1	j1	k1	l1	m1	n1	o1	p1	r1	Total probability of occurrences
Main Path (Reservation.)	1				2	3	4	5	6	7	8	9	10	11	12	13		*0.5*1*0.9*1*1* = **0.45**
Variant 1 (Normal served)		1		2	3	4	5	6	7	8	9	10	11	12	13	14		*0.3*1*0.99*1*1 *0.9*1* = 0.24057
Variant 2 (Carryout)			1	2	3		4	5	6	7	8	9	10	11	12	13		*0.2*1*0.1*1*1* = 0.02
Variant 3 (No Served)		1															2	*0.3*1*0.001*1* = 0.0003

Figure 10 (a). Action Matrix for Restaurant Service Application

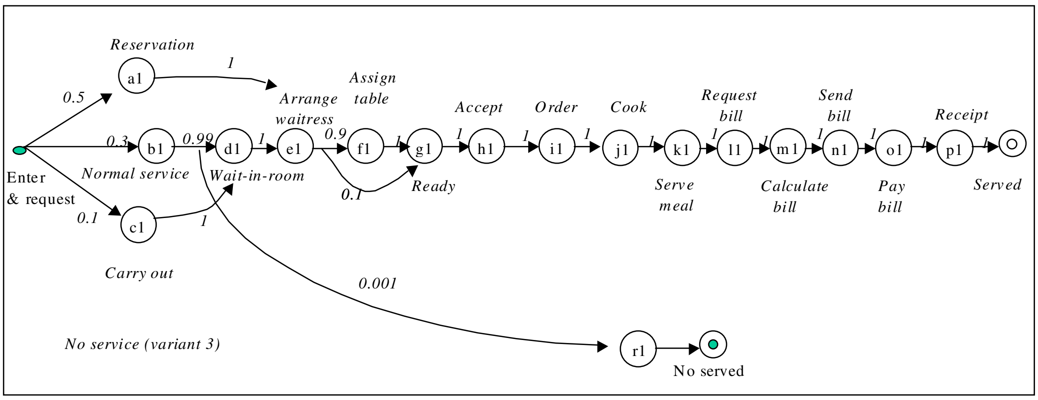


Figure 10 (b). Use Case Dialog Map for Restaurant Service Application

4. A Conversion algorithm from sequential diagrams to design unit/ Action unit

Hurlburt [5] provided a tabular approach, termed an action matrix, to the specifications of functional units. But, Hurlbut failed to provide a mechanism that generates action matrices from interaction diagrams. We have adapted his approach to clearly represent all possible scenarios without keeping too much duplication of action units. We also use the action matrix and the use case dialog map to organize the behavioral properties of the design, and to help generate the preliminary test scheme and test plan by analyzing the interaction diagram at the design stage.

The proposed algorithm has been influenced by several other researches. In this process we have extended the action matrix and use case dialog maps based on the advantages of both the Moore [8] and Mealy [7] models in that the Mealy model more easily represents a finite-state model by a table and the Moore model is a node-weighted finite-state model. In addition to applying these two models, we integrate Musa's idea of operation profiles into the refined action model and use case dialog map, which helps develop our test plan matrix approach [12,13].

The following assumptions have been used in the construction of the algorithm:

1. The input of the conversion algorithm is the basic interaction diagram(s) with either a single control object or multiple control objects.
2. The output of the algorithm is a collection of design units of which the action matrix consists. When there are no more external messages, the process is terminated.
3.
4. When an actor sends an initial external event to the system (that is, the interface object), it initializes a state S0.
5. Messages are called internal messages within the system (from the interface object), Messages can be events, methods (functions), or actions.
6. It happens the state changes on just the control object(s), and has the choice notation(s) which makes the decision (can apply n-ary branches) on the control object(s),
7. The state change on the control object(s)
8. For while internal messages are traced by the control object, they will be clustered by an action unit.
9. When a message exits the system or the interface object sends message to its own actor, the message is also called the external event.
10. Initialize indexes for identifying each message (m), clustering messages (i), setting to change state(j), and clustering dialogue(k).
11. On the choice notation, we traverse the interaction diagram(s) using Depth-First Search, called recursively.

How to identify design units to be traced on the Interaction Diagram? First, I explain a simple algorithm to identify design units in four steps:

1) Identify each *method unit* starting from the initial external event until there are no more messages on the interaction diagram.
2) Identify each *state* on the control object (the so-called 'state object'), which will change whenever a message comes in and out.
3) Also identify each *reusable patter unit* which is the cluster of sequenced methods based on the boundary of states.
4) Identify each *dialog unit* based on its response from the system after each external event occurs.

With the above steps, I developed an intermediate conversion algorithm which consists of three modules such as a main structural chart, two functional sub charts of left-DFS and right-DFS as follows:

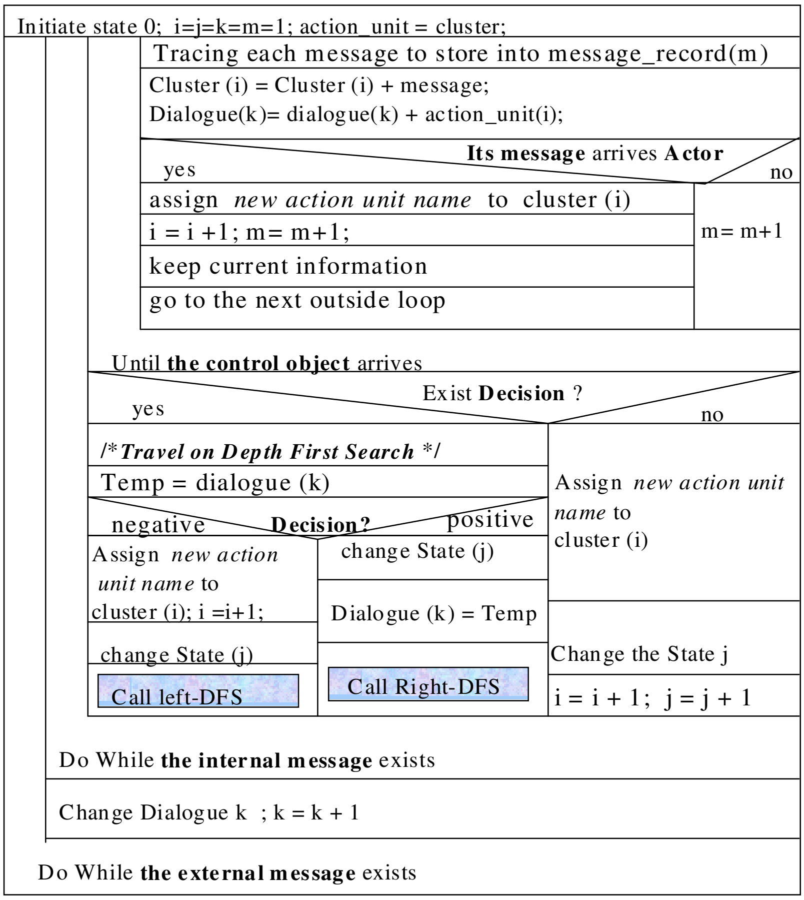


Figure 11 (a) The main structural chart of conversion alg.

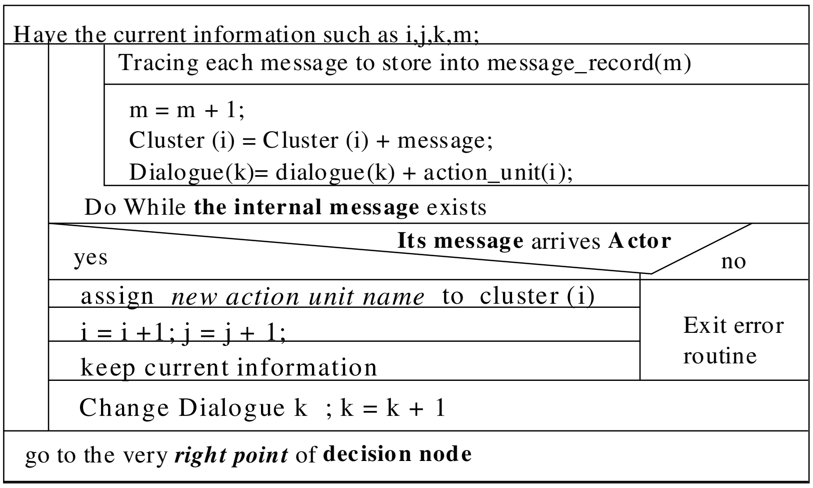


Figure11 (b) The Functional Sub-Structural chart of Left_DFS

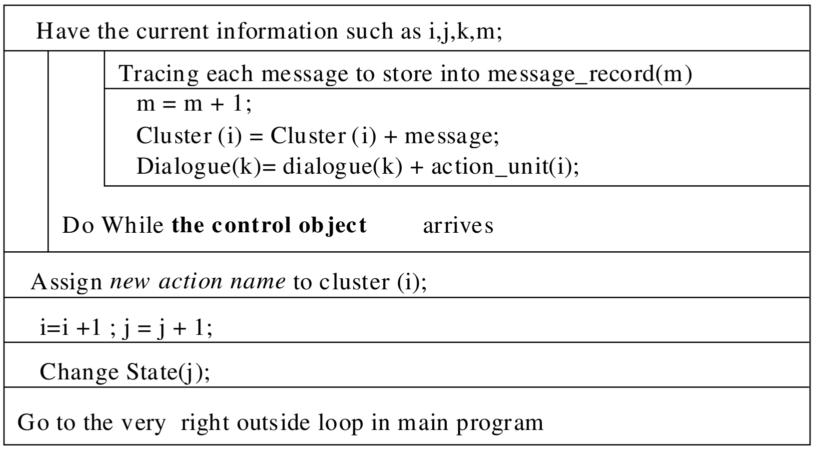


Figure 11(c) The Functional Sub-Structural chart of Right_DFS

5. Test Plan Metrics based on Design Stage

Most existing testing methods for white-box testing and black-box testing[1,11] are developed for traditional procedural programs. Recently, object-oriented software testing methods have been developed for object oriented programs with white-box testing. None of the existing methods of testing ‘design’ at the design stage can directly be applied for object oriented development methodologies. I propose to develop different metrics for test plan developed based design. My testing approach emphasizes testing software design specifications in the design stage while traditional software testers concentrate on testing the program source code. Actually, metric concepts are frequently used on routing in switching networks. But I brought the metric concepts to the software development in that the motivation for metrics should come from ordering the unit & acceptance tests due to the large numbers & ordering of scenarios in integration testing.

5.1. Test Plan Metrics

This chapter focuses on the software testing metrics used in the generation of object oriented test plans as part of Carlson’s use case methodology[3]. The test plan uses an action matrix that contains a collection of executable sequences of use case action units. The action matrix is generated from the interaction diagram at the design stage. Software testing metrics are employed to improve the productivity of the testing process through scenario prioritization. In other words, the software test metrics are available to evaluate the use case scenarios defined by the action matrix so that a test plan will emerge that improves the productivity of the testing process. The tester has a broad range of options to choose from when identifying ‘action units’. The simplest choice is to let each method be an action unit. A drawback to this approach is that the number of action units may be quite large. If so, the test plan generator can choose one of the other options based on the design unit (or test units) concept introduced. In selecting ‘reusable pattern units’, the assumption is that reusable pattern design units are also used as part of the design process. In selecting 'state units', the assumption is that an event state model has been developed. Presently, I have a conversion tool that analyzes interaction diagrams, and automatically generates 'state units'. The same tool can be used to automatically identify 'dialogue units'. The basic reason for choosing one of these units is that each offers its own unique approach to unit testing based on proven testing techniques. For example, reusable pattern test plans can be used with reusable pattern. State based testing techniques can be used with state units. Actor based acceptance testing can be used with dialogue units. Scenario based integration testing techniques can be used with use case scenarios to identify an ordered list of test scenarios. User acceptance testing can be used with use case requirements. Based on this choice, the test plan contains a set of action units together with appropriate unit testing techniques to be applied to these units. The software metrics described in the next section can be used to yield a more productive order in which these units can be tested.

The purpose of this chapter is to 'optimize' the order in which the scenarios defined by the rows of the action matrix are executed. This approach was adopted from Musa's work on Operational Profiles [9,10]. Musa's approach assumes that the designer has sufficient insight to assess the 'criticality' of action units and assign weighting factors to the elements of the action matrix [7,8]. This approach differs in that the designer analyzes the scenarios based on the 'reusability' of their components or subpaths.

Table 1 illustrates the test plan metrics such as most critical scenarios, most reusable components, and most reusable subpaths. The software test metrics described in this section focus on the length, criticality and reusability properties of the scenarios / action units as summarized in Table 5.1.

First, the issue of *Length* is two aspects of shortest path and longest path. I think it is not important for software design development. But it is useful if we use this issue with other categories of the metrics.

Second, the issue of *Criticality* is important to choose an ordered list of test scenarios.

Third, the issue of *Reusability* is also important to identify and maximize the reusable components. Therefore, we use scenarios and action units to develop a new path (i.e., scenario) with the smallest number of alterations from the existing paths.

To apply test plan metrics for each of the approaches described in Table 5.1 will be applied to the "Restaurant service" application.

We calculate total probability of occurrence as follows:

$\forall_i$ Use case **Scenario** $_i$ $\subseteq$ A Use Case **R** (**R** is Restaurant Application)

For all use case scenarios between the starting point and the ending point, the particular scenario **Scenario** $_i$ is included in a Use case **R**.

$\forall_i$ **action unit** $_i$ $\subseteq$ use case **Scenario** $_i$

For all action units within a particular use case **Scenario** $_i$ we can calculate the total probability of occurrence with ($\prod$ the weighed factor of Action unit $_i$ * probability of action unit $_i$) / ($\sum$ probability $_i$).

	Measures of test path		Weight value (w)
Length	1) Shortest path (simple path) - least steps of actions		**w = 1**
	2) Longest path (hardest path) - most steps of actions		
Criticality	**1) Most critical path**		**w≥ 1**
	2) Least critical path		**w≥ 0**
Reusability	**Component**	1) **Most reusable components**	**w > 1**
		2) Least reusable components	**w≥ 0 AND ⌉= 1**
	Sub-path	**Most reusable sub-path**	**w > 1**

Table 1. Test Plan Metrics

w: weight value ⌉: not

Figure 10 (b) shows the alternative representation of the action matrix, the use case dialog map, to apply the calculation of the total probability of occurrence in each use case scenarios. Figure 10 (a) shows tabularly all possible action units of each use case scenario in the use case restaurant service. The Mealy model and the Moore model are theoretically equivalent, but the Mealy model is a link-weighted model and the Moore model is a node weighted model [Beiz95]. We apply with both weight concepts. As a result, each action unit is assigned a weighted value with the value one and each link is also a probability of occurrence.

Most Critical Scenario

The first metric is an adaptation of Musa's 'most critical operational profile' approach [9,10]. This metric places greater weight on those scenarios that use action units thought to be most critical. It assumes that the designer can make these judgments. Later metrics will not have to assume that someone is available to make such judgments, since they can be produced automatically.
Figure 10 (a). shows the action matrix of the restaurant service application. The use case scenarios defined by the rows of this matrix include: reservation (variant 0), normal service (variant 1), carryout (variant 2), and no served (variant 3) use case scenarios. The probability of occurrence of each scenario is: variant 0 (0.45), variant 1(0.2673), variant 2 (0.02), and variant 3 (0.0003). As this result, we can make a decision to choose the 'reservation' scenario because it has the highest value of probability of occurrence. Figure 11 displays the ordered list of test scenarios as follows: the first direct path of reservation scenario which consists of the sequence of action units 'a1->e1->f1->g1->h1->i1->j1->k1->l1->m1->n1->o1->p1' with the amounts of weighted values equal to 13,

the second direct path of normal service scenario which consists of sequences of action units 'b1->d1->e1->f1->g1->h1->i1->j1->k1->l1->m1->n1->o1->p1' with the amounts of weighted values equal to 14, the third direct path of carryout service scenario which consists of sequence of action units 'c1->d1->e1->g1->h1->i1->j1->k1->l1->m1->n1->o1->p1' with the amounts of weighted values equal to 13, and the fourth direct path of no service scenario which consists of sequence of action units 'b1->r1, and other combinations like Figure 11.

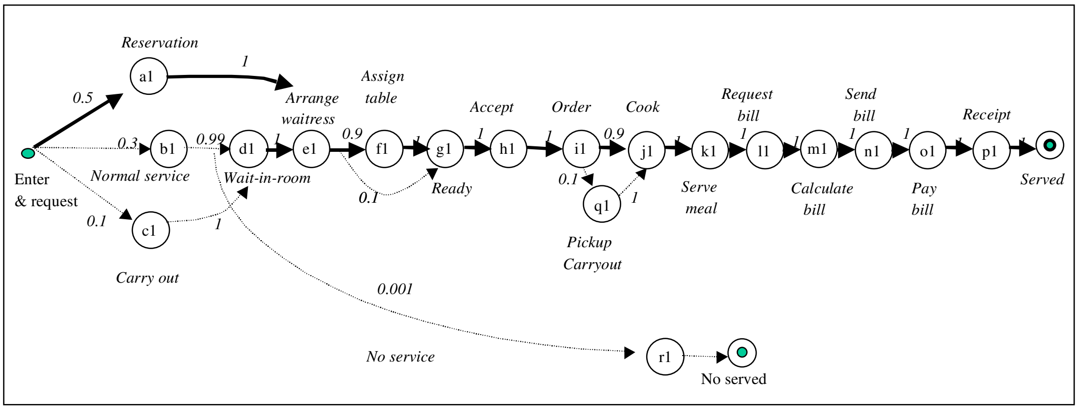


Figure 11. Ordered List of Test Scenarios

Most Reusable Components

This approach simply measures the reusability of action units in each row of the action matrix. This metric places greater weight on those action units that are reused the most by the collective group of scenarios being analyzed.

Figure 12 (a) displays three different types of geometric figures: a triangle, a rounded rectangle, and an oval. The triangle implies a particular component is used just one time on just a single one of the paths. The rounded rectangle implies that this component us used on two paths. The oval implies that this component is used on three paths. The reusability weight is defined as the number of paths that use the particular component.

Therefore, Figure 12 (b) shows the values 'reusability weight' of each action unit. The values can indicate whether a particular action unit is reusable or not. We may say that the unit action is reusable when the value of the particular unit is at least 2.

Figure 12 (c) indicates the total values of reusability components on each path (scenario). Due to the 'most critical scenario', we say that path1 (reservation) is better than path 2 (normal service), which is better than path 3 (carryout), which in turn is better than path 4 (no service). But if we measure each path based on the 'most reusable component', then we recognize that path 2 (normal service) is more usable than path 1 (reservation).

	a1	b1	c1	d1	e1	f1	g1	h1	i1	j1	k1	l1	m1	n1	o1	p1	r1
Main Path (Reservation.)	1				2	3	4	5	6	7	8	9	10	11	12	13	
Variant 1 (Normal served)		1		2	3	4	5	6	7	8	9	10	11	12	13	14	
Variant 2 (Carryout)			1	2	3		4	5	6	7	8	9	10	11	12	13	
Variant 3 (No Served)		1															2

(a)

	a1	b1	c1	d1	e1	f1	g1	h1	i1	j1	k1	l1	m1	n1	o1	p1	r1
Weighted value	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Reusability Weighty	~~1~~	2	~~1~~	2	3	2	3	3	3	3	3	3	3	3	3	3	~~1~~

(b)

	Path 1	Path 2	Path 3	Path 4
#3	11	11	11	0
#2	1	3	1	1
Total amount	35	39	35	2

(c)

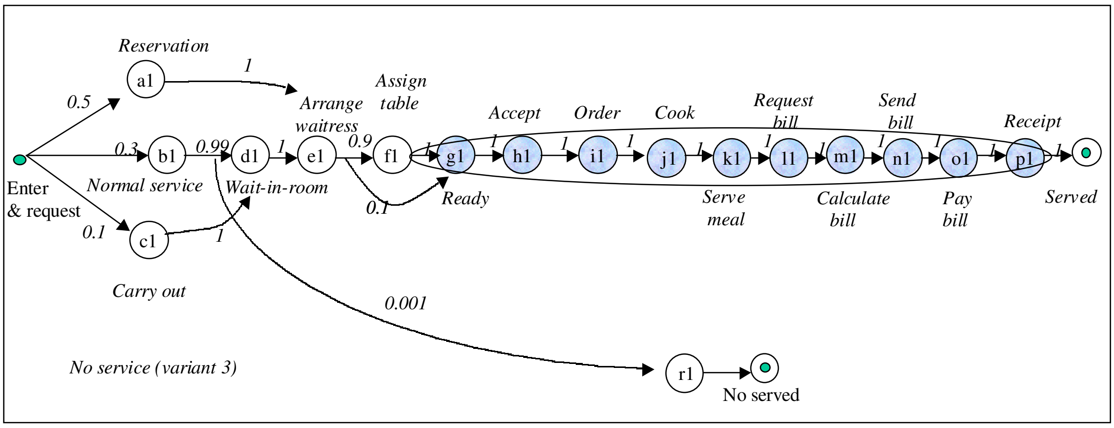
Figure 12. Most Reusable Component

Most Reusable Sub-Paths

This metric is similar to the previous metric except that it places greater weight on scenarios which share common subpaths. Figure 5.4 shows how to identify each cluster of the sequence of reusable components in all possible scenarios of the restaurant use case application. Figure 13 (a) shows three different types of geometric figures: an elliptical figure, a shaded elliptical figure, and a rounded rectangle. The elliptical figure shows the cluster over two paths with reusable subpaths. The shaded elliptical figures show iteratively or repeatedly the cluster of reusable subpaths in paths. The rounded rectangle displays the smallest cluster, which consists of two components, in paths, but it is less useful because this size is smaller than the smallest dialog unit within this application. Figure 13 (b) displays the core pattern (cluster) in the use case dialog map. On path1 and path2, we can see the 'longest reusable subpath' which is 'e1' through 'p1' represented by the ellipse. On path1, path2, and path3, we can see the reusable subpath 'g1' through 'p1', represented by the shaded elliptical figures.

	a1	b1	c1	d1	e1	f1	g1	h1	i1	j1	k1	l1	m1	n1	o1	p1	r1	**Total probability of occurrences**
Main Path (Reservation.)	1				2	3	4	5	6	7	8	9	10	11	12	13		*0.5*1*0..9*1*1*1* = **0.45**
Variant 1 (Normal served)		1		2	3	4	5	6	7	8	9	10	11	12	13	14		*0.3*1*0.99*1*0.9*1 *1*1*= **0.2673**
Variant 2 (Carryout)			1	2	3		4	5	6	7	8	9	10	11	12	13		*0.2*1*0.1*1*1*1* = **0.02**
Variant 3 (No Served)		1															2	*0.3*1*0.001*1* = **0.0003**

(a)



(b)

Figure 13. Most Reusable Subpaths

Length of Path

This metric is not meaningful for itself in our approach, but applies the most reusable subpath to measure the length of the subpath. Based on the ordered list of test scenarios with the most critical scenario, we can find the shortest and longest length of all paths from the starting point and the ending point.

	a1	b1	c1	d1	e1	f1	g1	h1	i1	j1	k1	l1	m1	n1	o1	p1	r1	**Weighted values**	**Total probability of occurrences**
Main Path (Reservation.)	1				2	3	4	5	6	7	8	9	10	11	12	13		**13**	*0.5*1*0..9*1* 1*1*= **0.45**
Variant 1 (Normal served)		1		2	3	4	5	6	7	8	9	10	11	12	13	14		**14**	*0.3*1*0.9.9*1 *0.9* 1*1*1* = **0.2673**
Variant 2 (Carryout)			1	2	3		4	5	6	7	8	9	10	11	12	13		**13**	*0.2*1*0.1*1* 1*1*= **0.02**
Variant 3 (No Served)		1															2	**2**	*0.3*1*0.001*1* = **0.0003**

Figure 14. Length of Path with Most Critical Scenario

If we just use this metric to identify a important path, it is not meaningful that path 4 is the shortest path with two action units and path 2 is the longest path with fourteen action units in Figure 5.4. In reality, we say that the shortest path (no service) is a dummy path (no service). After done by most critical scenario, we had better apply this metric to recognize the most important subpath. Therefore, we can use this metric of the shortest
and the longest path on the concepts of most critical scenario and most reusable component.

<u>Collection of Total Reusable Weight Values and Total Critical Weight Values</u>. Finally, we can calculate the collection of total reusable values and total critical weight values to find the significant unit/path. We adapt Musa's frequent weighted approach [Musa92,93]to our critical weighted issue.
We calculate total weighted values with the formula below:
Internal shared path (ISP) =

$$\sum \text{(Center-weight (CW) + Westside-weighted (WW) + Eastside-weighted (EW))}.$$

	a1	b1	c1	d1	e1	f1	g1	h1	i1	j1	k1	l1	m1	n1	o1	p1	r1	**Total Weighted values**	Total probability of occurrences
Main Path (Reservation.)	1	0	0	0	2	3	3	3	3	3	3	3	3	3	3	2	0	**35**	*0.5*1*0.9*1*1*1*= **0.45**
Variant 1 (Normal served)	0	1	0	2	3	3	3	3	3	3	3	3	3	3	3	2	0	**38**	*0.3*1*0.99*1*0.9*1*1*1*= **0.2673**
Variant 2 (Carryout)	0	0	2	3	2	0	2	3	3	3	3	3	3	3	3	2	0	**33**	*0.2*1*0.1*1*1*1*= **0.02**
Variant 3 (No Served)	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	**2**	*0.3*1*0.001*1* = **0.0003**

Figure 15. Collection of Total Reusable Weight and Total Critical Weight Values

I get the same result as the issue of the most reusable component in Figure 15 As a result, we can clearly determine a basic main path, by first making an ordered list of all paths.

6. Conclusion

Traditional software testers concentrate on testing the *program source code* in the implementation stage or testing stage, while our testing approach will emphasize testing software design specification (action matrix or state transition diagram/table) in the design stage. In the design phase of our object oriented development methodology, I will focus on testing 'action matrix & use case dialog maps' that is generated from sequential diagram (called interaction diagram), which represents the behavioral properties of system design. That is actually testing "specifications" before implementing real program source codes (program statements). We shall suggest scenario based testing which is consisted of *most critical scenario, most reusable components, and most* *reusable subpaths.* Scenario based testing will make a decision to order of all possible scenarios to test first, to maximize reusability, and to minimize test cases. As a result, this can lead for designer to design better system with information of reusable design components

References

[1] Breizer, Boris, “Black-Box Testing”, John Wiley & Sons, Inc, NY, 1995
[2] Byun, k. Use case based approach to algorithm event state table, Ph.D, Thesis Illinois Institute of Technology, Chicago, IL 1999
[3] Carlson, C.R.“Object-Oriented Information Systems: Architectural Strategies”, Viking Technologies Inc., Chicago, 1997.
[4] Firesmith, D. “Use cases: The Pros and Cons,” ROAD, Vol.2,No2,pp2-6, 1995.
[5] Hurlbut, R. “Managing Domain Architecture Evolution though Adaptive Use Case and Business Rule Models”, PH.D Thesis, Illinois Institute of Technology,1998.
[6] Jacobson, I., et al, “Objet-Oriented Software Engineering: A Use Case Driven Approach”, Addison-Wesley/ACM press,1992.
[7] Mealy, G.H. “A Method for Synthesizing Sequential Circuits”, Bell System Technical Journal Vol 34, 1955.
[8] Moore, E. F. “Gedanken Experiments on Sequential Machines”, In Automata Studies. Annals of Mathematical Studies #34. Princeton,Nj: Princeton University Press, 1956.
[9] Musa, J.D. “The Operational Profile in Software Reliability Engineering: An Overview”, AT&T Bell Labs. NJ,1992.
[10] Musa, J.D. “Operational Profile in Software Reliability Engineering: An Overview”, AT&T Bell Labs. NJ,1993.
[11] Marick, Brian, “The Craft of software testing: subsystem testing including Object-Based and Object-Oriented testing”, Prentice Hall Series, NJ, 1995.
[12] Kim, YoungChul, Carlson, C.R. “Scenario based integration testing for Object-oriented software development”, IEEE The Eighth Asian Test Symposium (ATS’99), November 16-18, 1999, Shanghai, China.
[13] Kim, YoungChul, Carlson, C.R "Adative Design Based Testing for OO Software", ISCA 15th International Conference on Computers and Their Applications (CATA-2000), New Orleans, Louisiana, March 2000