Chapter 4

Managing Test Scenarios

UML diagrams capture functionality of a system at various levels and can be used for activities like scenario generation, scenario prioritization and scenario selection. Dynamic diagrams are used to capture scenarios related to a use case. UML activity diagrams capture functionality in a form understandable to all the stakeholders of a system. UML use case and activity diagrams can be used for automated generation of scenarios. Scenarios thus generated are exhaustive and given constraints of cost and time, they can be prioritized or a subset of the same selected for execution.

4.1 Introduction

Requirements of a system are captured through use case diagrams, with each use case ideally, representing a functionality of the system. Usage scenarios describe the way a system will be used by actors. UML activity diagrams capture logic related to a single use case i.e. they elaborate a use case by specifying the order of activities, called scenarios. Thus, use cases along with corresponding activity diagrams describe expected behaviour of a system. Further, in the process of development, activities are implemented through design diagrams leading to program implementation[BS05]. So, activity diagrams are considered for testing of a system to ensure all user requirements are implemented without any fault.
Stakeholders can participate in testing (say, for acceptance testing) as activity
diagrams are well-understood by them. A scenario i.e. an activity path stretching
from start activity to end activity can also be considered as a test scenario during
the process of testing system requirements. The challenge testers face is the selec-
tion of test scenarios such that testing of requirements is maximized with higher
coverage and lesser effort[RH96].

Test scenario management involves a gammut of techniques, namely, test sce-
nario(case)\(^1\) generation, test case prioritization, test case selection, test suite gen-
eration, test data generation, developing a test oracle as well as generating reports
detailing test case execution progress. This chapter discusses techniques for gen-
eration, prioritization and selection of scenarios obtained from UML diagrams,
specifically, use case and activity diagrams. Here, it is assumed that the models
have been checked for consistency which has been discussed in the previous chap-
ter. A test suite is a composition of tests i.e. it runs a collection of test cases.
Scenarios obtained from UML diagrams can be used to compose a test suite.

Test scenario generation involves enumerating all possible scenarios from one
or more UML models for testing. The number of scenarios generated being large,
it is not feasible to test exhaustively given constraints of time, effort and cost.
This work introduces techniques to reduce the number of scenarios generated,
specifically, in case of concurrent activities using priority/level of activities as the
basis(Section 4.2).

The order of execution of test scenarios affects the time at which objectives
of testing are met. For example, if maximizing coverage is the objective, an
inappropriate execution order would take a large number of tests to achieve the
goal. Focus of Section 4.3 is prioritization, which involves ordering test cases in
a test suite to achieve a test objective. Execution of scenarios in random does

\(^1\)Test cases are derived from scenarios. Thus, a scenario consists of one or more test cases.
Hence, test scenarios and test case are used synonymously here.
not always produce best results. Therefore, it is necessary to select a subset of all possible scenarios that best meet an objective (e.g. fault detection, coverage). Existing work looks at factors like customer inputs, nature of requirements and risk associated as the basis for prioritization. In this work, primitives of use case and activity diagrams are used to evolve a measure to aid prioritization of use cases and scenarios.

Exhaustive testing being impossible, it is necessary to find a subset of scenarios that best represents the entire set of scenarios for testing. This is especially crucial in case of testing given limited resources and time. Test selection techniques aim at selecting a representative subset of test scenarios from the total set of scenarios for testing (Section 4.4). Levenshtein distance is used to calculate the distance between scenarios. A second technique adapts the idea of Longest Common Substring [Mai78]. Subscenarios, their length and position in the sequence is taken into consideration to calculate distance between scenarios. A third technique introduced in this work uses clustering (Agglomerative clustering) for grouping scenarios based on a distance metric. Clusters group similar scenarios together, thereby aiding the process of selection.

Random selection of scenarios for selection is not feasible as they may select similar scenarios for testing [CLM04]. Risk, coverage, cost and efficiency are used as factors to select scenarios in [BLFR02, CP03a]. Values for risk and cost are provided by the users of the system and hence require user intervention. The techniques are advantageous in terms of user inputs to ascertain scenarios most important to the customer. However, they lack in ease of use. This work looks at using distance measures calculated between scenarios as the basis for selection.

In this chapter, techniques to manage test scenarios are presented. Section 4.2 presents techniques to reduce the number of scenarios generated in case of concurrent activities. Section 4.3 presents techniques for prioritizing use cases (Section 4.3.6)
and scenarios (Section 4.3.8). Techniques for scenario selection is the focus of Section 4.4.

4.2 Generating scenarios from UML Activity Diagrams

4.2.1 Introduction

Specification-based testing, also called black-box testing, involves producing a test suite based on the specification. Using a formal language or a model for specification helps in automation of the test generation process. For large and complex systems, testing based on covering the control flow or data flow paths becomes infeasible. In this regard, an efficient set of test scenarios needs to be generated. One of the main objectives of testing is to check whether customer requirements are met. Scenarios help in generating sequence of activities that implement system objectives/requirements.

Requirements are well defined using activity diagrams and this has led to an increased interest on generating test scenarios using activity diagrams [KKBK07, CMK08]. Each path from the initial node to the final node in an activity diagram constitutes a scenario. The problem encountered following the strategy is exponential increase in scenarios especially when considering concurrent activities, represented in an activity diagram using fork-join nodes [BL01b, LL05c, KKBK07, XLL07]. It is observed that the growth in scenarios can be limited by considering domain dependency existing among concurrent activities\(^2\) which is the focus of this section. A method is proposed to minimize scenarios generated in case of concurrent activities, to aid the scenario generation process.

\(^2\)A part of the work stated here is published in the Proceedings of the 11th International Conference on Information Technology, 2008 and appears in IEEE Xplore Digital Library
Section 4.2.2 and 4.2.3 discusses the selection of activity diagrams to represent scenarios. Existing work in the area of scenario generation is discussed in Section 4.2.4. The approach used in this work to generate scenarios in case of concurrent activities is discussed in Section 4.2.5. Section 4.2.6 discusses coverage criteria to evaluate the technique and Section 4.2.7 summarizes the work.

4.2.2 Why Activity Diagrams?

A use case capturing a requirement consists of scenarios i.e. each scenario can be said to represent a requirement goal of the system. Scenarios thus represent the sequence of events in a software system and defines a system’s behaviour. A scenario has a defined goal, begin with a triggering event and end by either successful or unsuccessful completion of the task. In practice, generation of scenarios is mostly done manually making it labor-intensive and error-prone. Hence, there is a need to generate test scenarios to achieve test adequacy and to ensure software quality [Mat07]. Work in literature use activity, sequence/collaboration diagrams to represent scenarios [AO00, FL02, MXX06]. In this regard, automation of test scenario generation gains importance.

The survey by Dobing et al [DP08] on the use of UML diagrams show that UML diagrams (use case narratives, use case diagrams, sequence diagram and activity diagrams) find more use in case of organizations that are middle sized to large, employing an average of more than 50 employees, the size of the project and organizational usage. Another important factor affecting the same is availability of UML tools.

Work is available in literature wherein scenarios represented using sequence diagrams are used for test case generation. Frainkin et al [FL02] have developed SeDiTeC, a tool that supports automated testing based on testable sequence diagrams. Technique for testing polymorphic interactions defined in sequence di-
agrams is presented by Suravita [SS05]. Cengarle et al [CGW06] introduce two operators, 'variant' and 'repeat' to represent variability in interactions. They use 'variant' to represent optional behaviour and 'repeat' for repetition of instances. An algorithm is defined by Lund et al [LS06] for deriving tests from sequence diagram specifications that use operators 'neg' and 'assert' that represent invalid and universal behavior. Use case diagrams, corresponding sequence diagrams, class diagrams and OCL is used by Briand et al [BL02] for specification based testing. They introduce a methodology to build functional system requirements which is transformed to test cases, oracles and drivers once detailed design information is obtained.

Sequence diagrams are used to represent scenarios, especially, in the design phase. The objective is to find architectural, interface and logic problems as the sequence diagram gives details about interfaces, states, message order, assignment of responsibilities, timers and error situations. Besides, sequence diagrams are a useful tool for specifying system requirements for programmers as they clarify understanding of the application among technical members of the project team [DP08]. Thus, while being an effective tool for use by developers, the drawback of sequence diagram is in terms of understandability by different stakeholders of the system. Compared to the sequence diagram, activity diagrams find better acceptance among stakeholders in terms of customer involvement through verification and validation of requirements. The advantage lies in its simplicity and the ease of understanding the flow of logic of the system. Also, activity diagrams can be used at different levels of abstraction aiding both the customers as well as developers in capturing requirements[OMG].

Another development is the change brought in the UML 2 superstructure specification [OMG] wherein there is a clear delineation of activity diagrams from state diagrams. Also, non-availability of tools and techniques related to activity dia-
grams have been an impediment. The Object Management Group (OMG) [OMG] also classifies activity diagrams as Fundamental, Basic, Intermediate, Structured and Complete in terms of complexity in the process flow. The basic level includes control sequencing and data flow between actions. However, forks and joins as well as decisions and merges are not supported. The intermediate level supports concurrent control and data flow, and decisions. The complete level adds constructs that enhance the lower level models, such as edge weights and streaming. This work is concerned with the intermediate level of activity diagrams that include control and data flow, and decisions. Scenarios are generated from activity diagrams for the purpose of testing.

4.2.3 Activity Diagrams and Scenarios

An activity diagram represents the scenarios for each use case in a use case diagram [CPTT05]. UML activity diagrams are developed using two types of nodes, namely, action nodes and control nodes as shown in Figure 4.1. Action nodes include Activity, CallBehaviourAction, SendSignal and AcceptEvent. Control nodes include InitialNode, FinalNode, FlowFinal, Decision, Merge, Fork and Join. Also, UML 2.1 superstructure describes several levels of activity modeling: Basic, Intermediate, Complete, Structured, complete-Structured and Extra-Structured activities [OMG] as discussed in the previous section.

An activity diagram consists of activities and transitions, showing the flow of control from one activity to another. They can be used to model both sequential and concurrent activities. Also, an activity diagram can be viewed as a graph with nodes representing activities and edges labeled with transitions. Figure 4.2 shows an activity diagram for booking of a package tour by a customer. The activities inside a fork-join are concurrent activities and are executed in parallel. Other activities are sequential in nature.
Based on the above, the following definitions follow:

**Definition 1:** An activity diagram AD is a tuple,

$$AD = (A, T, F, J, R, a_I, a_F)$$

where

- $A = a_0, a_1, a_2,...a_m$ is a finite set of activities,
- Priority of an activity 'a' is denoted by $a.p$,
- $T = t_0, t_1, t_2,..., t_n$ is a finite set of transitions,
- $F = f_1, f_2, ..., f_k$ is a finite set of forks,
- $J = j_1, j_2, ..., j_k$ is a finite set of joins,
- $R = R \subseteq (A \times T)$
- $a_I$ is the initial state, and $a_F$ is the final state.

**Definition 2:** A scenario, $s \in S$ (S being the set of scenarios), in an activity diagram, AD, can be defined as an execution path from the initial state to the final state consisting of activities and transitions.

i.e. $\forall$ $s$, where $s \in S$,

$s = a_0 \rightarrow t_0 \rightarrow a_1 \rightarrow t_1 \rightarrow ... \rightarrow t_n \rightarrow a_m$

where $a_i \in A$, $t_i \in T$, $a_0$ is the initial state, $a_m$ is the final state and $S$ is the set of scenarios.
Figure 4.2: Activity diagram for "Booking a Package Tour"
4.2.4 Related Work on Scenario Generation using UML Activity Diagrams

Lionel et al [BL01b] in their work present an approach for UML-based testing using activity diagrams. They capture the sequence of usage scenarios related to the use cases. An activity diagram is transformed into a weighted graph from which the sequence of usage scenarios are identified. This helps in determining paths for testing. A modified Depth-First Traversal [DFS] algorithm is used for automated generation of test cases by Linzhang et al [LJX04]. The objective was to elucidate basic paths for the activity diagram. An advantage of the technique is that basic paths are explored but then it does not generate all possible scenarios which is a drawback. Also, their algorithm works on the assumption that each branch/merge and fork/join has only two outgoing edges. Li et al [LL05c] use ant-ant like agents to generate test threads from UML activity diagrams. However, path explosion problem is the limitation of this work as it results in redundant exploration of the activity diagrams. Adaptive agents are used for test scenario generation by Xu et al [XLL05] to overcome the problem of path explosion. The algorithm uses one agent at the start and produces more agents when necessary. The assumption is that each cyclic loop is executed at most two times, similar to the assumption in [LJX04]. They overcome the problem of path explosion in [LL05c] by creating agents only where necessary.

In order to derive test cases from UML activity diagrams, Kim et al [KKBK07] in their work convert the activity diagram into an Input/Output explicit Activity Diagram (IOAD). IOAD is an activity diagram that explicitly shows external inputs and outputs. The IOAD is then converted to a directed graph for extraction of test scenarios. Basic path is the coverage criteria. This work also looks at simple fork-join structures.

Xu et al [XLL07] in their work, overcome the limitation in previous work (i.e.
<table>
<thead>
<tr>
<th>Reference#</th>
<th>Technique</th>
<th>Limitation</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BL02]</td>
<td>- ADs capture sequence of usage scenarios related to use cases</td>
<td>Transformation to weighted graph</td>
<td>TOTEM</td>
</tr>
<tr>
<td>[LJX+04]</td>
<td>- Modified DFS</td>
<td>Loops executed atmost once</td>
<td>UMLTGF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Branch/Merge, Fork/Join have atmost two outgoing edges</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Some scenarios not generated, like fork-join</td>
<td></td>
</tr>
<tr>
<td>[CLL05]</td>
<td>- Modified DFS</td>
<td>Scenario explosion in case of loops</td>
<td>AD2US</td>
</tr>
<tr>
<td></td>
<td>Poseidon 2.1 used for modelling</td>
<td>- Handles simple fork-join constructs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Sequentially connected execution paths between fork/join</td>
<td></td>
</tr>
<tr>
<td>[LL05c]</td>
<td>- Anti-ant like agents</td>
<td>Scenario explosion in case of loops</td>
<td>A(unkwn)</td>
</tr>
<tr>
<td></td>
<td>- Redundant exploration of threads avoided</td>
<td>- Handles simple fork-join constructs</td>
<td></td>
</tr>
<tr>
<td>[XLL05]</td>
<td>- Uses Adaptive agents</td>
<td>- Each nested fork-join considered as AD</td>
<td>TSGAD</td>
</tr>
<tr>
<td></td>
<td>- Agents produced when required</td>
<td>- Same as fork-join for branch-merge</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Redundant exploration of threads avoided</td>
<td>- Overhead of merging ADs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Mixed fork joins not considered</td>
<td></td>
</tr>
</tbody>
</table>

AD - Activity Diagram; NA - Not available; A(unkwn) - tool present, name unknown.
## Testing Approaches using Activity Diagrams

<table>
<thead>
<tr>
<th>Reference#</th>
<th>Technique</th>
<th>Limitation</th>
<th>Tool</th>
</tr>
</thead>
</table>
| [CMX06], [CQX+07b] | -Objective to meet coverage criteria  
  -Generates random test cases from JAVA program  
  -Generates program execution traces  
  -Compare traces with activity diagram to achieve coverage criteria | -Overhead of generating random test cases  
  -Technique is exhaustive | AGTCG |
| [XLL07] | -Handles complex fork/join pairs | -Does not consider other constructs of AD | TSUMLAD |
| [KKBK07] | -Input/Output Activity Diagrams(IOAD)  
  -Transformed to directed graph | -Specifying input/output of each activity | NA |
| [CMK08] | -ADs translated to NuSMV, a formal model  
  -Properties as CTL/LTL formulas | -Additional overhead in converting ADs to NuSMV | A(Unkwn) |
| [XLLP08] | -Handles exceptions, expansion and interruptible activity regions | | TGUML |

AD - Activity Diagram; NA - Not available; A(Unkwn) - tool present, name unknown
simple fork-join) by considering complicated fork-join pairs i.e. fork-join pairs that are nested or those that have loops and branches. A prototype tool, TSUMLAD, was developed to automatically generate test scenarios from activity diagrams. All possible scenarios are generated by this technique. However, the number of scenarios thus generated is very large and it may not be possible to test all scenarios given constraints of resource and time. Table 4.1 compares and summarizes the various approaches to generating scenarios from UML activity diagrams.

Work in literature look at various ways of generating scenarios. However, one common issue faced is scenario explosion especially in case of fork-join constructs. Lionel et al [BL01b] and Linzhang et al [LJX] handle this issue by constraining the number of outgoing edges a fork/join construct can have. Xu et al propose algorithms to generate all possible scenarios in case of complicated for/join constructs. While the former constrains the representation of requirements, the latter creates the problem of scenario explosion.

To overcome the problem of scenario explosion, there is a need to generate scenarios in an optimized way, especially in case of concurrent activities. This work concentrates on generation of scenarios in case of concurrent activities represented using fork-join constructs by taking precedence of activities into consideration. It is done in two ways: one, using priority of activities, obtained by giving an order in which activities are generated and two, by defining levels, wherein activities at one level have higher precedence over activities on another level\(^3\). The methods are discussed in the next section.
Table 4.2: Scenarios from activity diagram ‘Book Package Tour’

| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₅ → a₆ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₅ → a₆ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₅ → a₆ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₅ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₆ → a₅ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₅ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₅ → a₆ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₅ → a₆ → a₄ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₆ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |
| a₀ → a₁ → a₂ → f₁ → a₃ → a₄ → a₅ → a₆ → j₁ → a₇ → a₈ → f₂ → a₉ → a₁₀ → j₂ → a₁1 |

4.2.5 Approach to Deriving Test Scenarios for Concurrent Activities inside Fork-Join

Concurrent activities in a system are represented using fork-join constructs. In case of concurrent activities, several activities can be active at the same time. They may share data in any order as the order of accessing is non-deterministic. Hence, such access of shared data may bring in unexpected results [LHS01].

Example: Consider the example of the use case, booking a package tour. The activity diagram of the use case is shown in Figure 4.2. The activities are named $a₀, a₁, ...,$, the transitions $t₀, t₁, ...,$, the forks/joins as $f₁, f₂, ...$, and $j₁, j₂, ...$, for ease of use. Note that in this work the details of each activity, i.e. actions are not

\textsuperscript{3}A part of the work stated here is reported in 'Automated Scenario Generation based on UML Activity Diagrams', 11th International Conference on Information Technology (ICIT 2008), pp. 209-214, IEEE Computer Society, 2008
Algorithm 1 ScenGen

1: { Input:
\hspace{1em} G: An activity graph obtained from activity diagram, AD.
Output:
\hspace{1em} Set of Scenarios
Functions called:
\hspace{1em} fork-join(G') returns all paths within a fork join construct which is a subgraph G'
Initialize:
\hspace{1em} Visited of each node = 0. Stack S to keep track of visited nodes.
\hspace{1em} SCEN stores scenarios. Adjacency - adjacency matrix that stores transitions between nodes present in the activity diagram. }

2: Begin

3: Traverse the graph using depth-first-search(DFS) from the initial nodes to the final node. For each node visited during the traversal, increment the number of visits.

4: If type of node = fork, then , get all nodes until corresponding join, forming a sub graph, and call function fork-join(G’) to generate all paths within fork-join construct. Set visited for each node. Return the set of subpaths.

5: If reached final node or number of visits of current node = 3, then store scenario. Backtrack to the node which has atleast a child node with number of visits less than two. Pop nodes until this point from S.

6: Perform above steps until all nodes are visited and final node is reached.

7: For each scenario in SCEN, check if the last node is a final node. If yes, print the scenario. Else, get all sub-paths from the last node to the final node and append to the scenario. Print scenarios

8: End
considered to preserve the simplicity of the activity diagram. $a_1$, $a_2$, $a_7$ and $a_8$ are non concurrent activities whereas $a_3$, $a_4$, $a_5$, $a_6$, $a_9$ and $a_{10}$ are concurrent activites. For a large and complex system, concurrent activities lead to path explosion due to a large number of threads that might be in operation at the same time. The order of execution of concurrent activities cannot be determined apriori as it depends on the runtime environment. Hence, the complexity in testing concurrent activities is $n!$ where $n$ is the number of activities within a fork-join construct. The steps involved in generating scenarios from an activity diagram is given in Algorithm 1. Depth first traversal is used to traverse the graph. Applying the algorithm $ScenGen(G)$ on the activity graph shown in Figure 4.2, the scenarios obtained are as shown in Table 4.2.

At first, customer details and trip details are obtained. Once the details are obtained, concurrent activities begin. Concurrent activities including booking the flight, hotel, cab and tour are performed. All of the four activities must be performed successfully for a trip to be booked. Also, the order in which the concurrent activities get executed is not predictable and can be in any order. Once, the four activities produce an output, the amount is computed and confirmation of the trip is taken from the customer. Payment is made towards the trip and again concurrent activities of 'generating invoice' and the 'tour folder' is performed. In this example, a simple fork-join construct is used in the activity diagram. A total of 48 scenarios can be generated considering the different ordering in calling concurrent activities. Thus, the number of scenarios generated in case of a concurrent activities is very large and impossible to test within given constraints.

One way to reduce the number of scenarios generated in case of concurrent activities is to discard illegal or irrelevant combinations of activities that may occur due to random selection. Consider the case in Figure 4.2. Suppose that the tour and cab booking activities, being in house can be arranged by the business in one
Figure 4.3: Interleaving of activities inside fork-join
of their establishments, dependent on the arrival of the customer and availability of accommodation. Then, this constraint in terms of order of execution could be used effectively in generation of scenarios. Figure 4.2 shows the interleaving of activities within the fork-join construct. The activity 'Book Hotel()' depends on confirmation from the activity 'Book Flight()'. In case the latter is not available, then the execution of the former is not applicable. Similar is the case of dependency between 'Book Hotel()' and 'Book Flight()' with activities, 'Book Cab()' and 'Book Tour()'. Thus, sharing of data among activities brings in dependency that enforces an order among concurrent activities; for dependency, only some sequences of activities are realistic and hence others may be discarded.

Figure 4.3 shows the case of 'Booking a Package Tour' with interleaving among the activities. The activity 'Book Flight()' sends a signal to 'Book Hotel()' once it is completed. This signal fires the activity 'Book Hotel()' to completion. The same is true of 'Book Cab()' and 'Book Tour()'. This dependency between activities can be used to generate a subset of scenarios thereby reducing the test effort. The scenarios generated based on the above is shown in Table 4.3.

Table 4.3: Test Scenarios for ‘Book Package Tour applying priority-based selection’

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_0 \rightarrow j_2 \rightarrow a_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow a_5 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow a_5 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_0 \rightarrow j_2 \rightarrow a_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this work, two criterion based on precedence are defined for the generation of scenarios in case of concurrent activities.

**Criterion 1: Priority-based selection**

Let priority of an activity $a_i$ be denoted as $a_i.p$. If the priority of $a_1$ is greater than the priority of $a_2$, then $a_1 \rightarrow a_2$ is a legal scenario.
i.e. \( \forall a_i, p_i > a_j, p_j \land a_i, a_j \in A \mid a_i \rightarrow a_j \in S \)

An algorithm \( \text{Priority-Based-ScenGen()} \) (Algorithm 2), is proposed to generate all scenarios satisfying the above criterion. The function is called when a fork node, \( f_i \) is encountered. All activities originating from the fork node and till the join node are considered. A queue \( Q \) contains all concurrent activities originating from the fork, \( f_i \), in order of priority. An adjacency matrix is used to store details on transitions between activities i.e. element \( AM[i][j] \) is set to 1 if there exists a transition from activity \( a_i \) to \( a_j \). Priority of each activity is stored in \( P \) and \( L \) stores the generated scenario. By applying Algorithm 2, a reduced set of test scenarios is obtained. Consider that the priority of activity Book_Flight() is greater than priority of activity Book_Hotel() whose priority is greater than the priority of the activities Book_Cab() and Book_Tour(); priority of activity Book_Cab() and Book_Tour() are equal. Then, the order of execution of concurrent activities is altered due to the priority they hold with respect to each other. That is, Book_Flight() having highest priority is executed first followed by activity Book_Tour() followed by Book_Cab() and Book_Tour(). Application of the algorithm to the activity diagram in Figure 4.3 gives the scenarios shown in Table 4.3. Thus, introducing priority for activities within the fork join construct helps reduce the number of scenarios generated. Once all the test scenarios are generated, test cases can be generated to form the test suite.

**Criterion 2: Level-based selection**

Activities inside a fork-join can be said to be at different levels of execution due to the dependency that exists between them. That is, activities at one level must be completed before moving to activities at the next level. Hence,
Algorithm 2 Priority-Based-ScenGen

1: \{ Input:
    \begin{itemize}
    \item $Q$: Concurrent activities ordered by priority originating from fork
    \item $AM$: Adjacency Matrix where $AM[i][j]=1$ if there exits a transition from activity $a_i$ to $a_j$.
    \item $S$: Generated Scenario
    \item $P$: Priority of each activity
    \end{itemize}
    Output:
    Scenarios
\}

2: while $Q$ not empty do
3:     Get activity $a$ from $Q$;
4:     if $a$ not in $S$ then
5:         Add $a$ to $S$;
6:     end if
7:     if all activities present in $S$ then
8:         Print $S$;
9:         exit();
10:    end if
11:    for each activity $a_i$ do
12:        if there exists a transition $t_i$ from $a$ to $a_i$ such that $P(a) > P(a_i)$ then
13:            set transition $t_i$ to visited
14:            if $a_i$ has no other activities with higher priority incident on it then
15:                set activity $a_i$ to visited
16:                add $a_i$ to $S$
17:            end if
18:        end if
19:    end for
20: end while
Let $L = \{ l_1, l_2, ..., l_n \}$ be a finite set of levels;

Let $\{ a_0, a_1, a_2, ..., a_j \} \in l_1,$

$\{ a_{j+1}, a_{j+2}, ..., a_{k} \} \in l_2, ....,$

then, the operation priority of all activities in a particular level is the same.

i.e. $a_0.p = a_1.p = ... = a_j.p.$

Also, $l_1.p > l_2.p.$ i.e. operation priority of activities in level $l_1,$ is greater than operation priority of activities in level $l_2.$

In case of criterion 2, all activities at level $l_i$ have to be completed before moving to activities at level $l_{i+1}.$ Procedure $LevelBasedGen$ gives the procedure for level-based selection. The level of each activity is stored in LVL. In case of nested fork-joins, the algorithm is called recursively.

Algorithm $Level-Based-ScenGen()$ (Algorithm 3) gives the steps involved in level based generation of scenarios. Consider a set of activities, where certain activities need to be executed(performed) before other tasks. This defines a precedence order. The precedence constraints form a directed acyclic graph. The approach defines an order to execute activities such that each activity is executed only after the activities incident on it are completed. Application of the algorithm to the activity diagram in Figure 4.3 gives the scenarios shown in Table 4.4. Introducing levels for execution of activities based on dependency among concurrent activities help reduce the number of scenarios generated.

<table>
<thead>
<tr>
<th>Table 4.4: Test Scenarios for 'Book Package Tour applying level-based selection'</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$</td>
</tr>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$</td>
</tr>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_0 \rightarrow j_2 \rightarrow a_{11}$</td>
</tr>
<tr>
<td>$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_9 \rightarrow j_2 \rightarrow a_{11}$</td>
</tr>
</tbody>
</table>
Figure 4.4: Interleaving of activities inside fork-join
Algorithm 3 Level-Based-ScenGen

1: \{ 
    Input:
    \( Q \): Concurrent activities originating from fork
    \( AM \): Adjacency Matrix where \( aM[i][j]=1 \) if there exits a transition from activity \( a_i \) to \( a_j \).
    \( S \): Generated Scenario
    \( Lvl \): Level of each activity
    Output:
    Scenarios
\}

2: while \( Q \) not empty  do
3:     Get activity \( a \) from \( Q \);
4:     Add \( a \) to \( S \);
5:     if all activities present in \( S \)  then
6:         Print \( S \);
7:         exit( );
8:     end if
9:     for each activity \( a_i \) do
10:        if there exists a transition \( t_i \) from \( a \) to \( a_i \) such that \( LVL(a) > LVL(a_i) \) then
11:            set transition \( t_i \) to visited
12:        if \( a_i \) has no other activities with higher level incident on it then
13:            set activity \( a_i \) to visited
14:            add \( a_i \) to \( S \)
15:        end if
16:     end if
17:     end for
18: end while
4.2.6 Test coverage criteria

Test quality is one of the key issues of the testing process. Measurement of test quality can be done by defining the adequacy criteria for testing[Mat07]. Adequacy for activity diagrams are based on covering the elements, namely, the activity states and transitions. Test assessment is a measurement of the goodness of the test set and is carried out based on one or more criteria, such as activity coverage, transition coverage, path coverage, condition coverage and loop coverage[kaw03].

Test coverage criteria is a set of rules that guide to decide appropriate elements to be covered to make test case design adequate[KK09b]. The following coverage criterion was used to consider efficacy of the technique:

a. Basic path coverage criterion

A basic path is a sequence of activities(scenario) where an activity in that path occurs exactly once[LJX+04, MXX06]. A basic path considers a loop to be executed at most once. Thus, given a set of scenarios, $S$, obtained from an activity graph and a set of test cases $TC$, for each scenario $s_i \in S$, there must be at least one test case $tc \in TC$ such that when system is executed with the test case $tc$, $s_i$ is exercised. For example, $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$ is a basic path.

b. Activity coverage

Activity coverage ensures that each activity in the activity diagram be covered at least once. Thus, given a set of scenarios, $S$, obtained from an activity graph and a set of test cases $TC$, for each activity $a_i \in G$, there must be at least one test case $tc \in TC$ such that when system is executed with the test case $tc$, $a_i$ is exercised.

c. Transition coverage

Transition coverage ensures that each transition in the activity diagram be covered at least once. Given a set of scenarios, $S$, obtained from an activity graph and a set
of test cases $TC$, for each transition $t_i \in G$, there must be at least one test case $tc \in TC$ such that when system is executed with the test case $tc$, $t_i$ is exercised.

### 4.2.7 Scenario Generation: Summary

The technique handles automated generation of test scenarios for concurrent activities. Concurrent activities, though conceptually concurrent are interleaved. This knowledge has been used to define dependency among the activities inside fork-join pairs. This information helps in generating those test scenarios that are possible, thereby reducing unwanted scenarios. Besides, the dependency information helps in design of specifications too. This approach has the following advantages:

a. Scenarios are generated based on dependency of activities

b. Only valid scenarios are generated thereby reducing the number of scenarios for which test cases have to be generated

c. It meets the activity, transition and path adequacy criteria

Generating scenarios using an automated approach from specification captured by activity diagrams has the advantage of being understandable by all stakeholders thereby providing a common platform. Besides, automated generation of scenarios aids the testing process as scenarios are available for testing early in the lifecycle. This has the added advantage of being a verification mechanism to check for ambiguity and mismatch between requirements too. The scenarios thus generated maybe used for testing directly. However, being ordered randomly, the test suite may not be effective in detecting defects or in achieving maximum coverage early. Prioritizing scenarios towards an objective like fault detection and coverage makes testing effective, techniques for which are discussed in Section 4.3. Also, exhaustive testing being impossible, there is need to select a subset of test scenarios that best
represents the entire set of scenarios. Techniques for test scenario selection is discussed in Section 4.4.
4.3 Prioritizing Scenarios

4.3.1 Introduction

Testing is a continuous process done across the software development life cycle with the objective of maximizing test performance like early defect detection or maximizing coverage. As software systems evolve, the size of the test suite grows making exhaustive testing infeasible. Constraints of cost, time and effort require that testing is done in an optimized way. To this end, prioritization helps in ordering test cases according to their importance.

Different indices have been used to prioritize scenarios and generally a single index is used for prioritization. Examples of prioritization indices are\cite{TAS06}:

- priority or criticality of requirements
- rate of fault detection
- level of coverage achieved by test case e.g. number of statements or branches
- complexity metrics e.g. cyclomatic complexity, computed for procedures executed by each test case
- historical information e.g. fault proneness

Customers are involved in a project to minimize the risk of misinterpreted and missing requirements\cite{Sri04, SWO05, SW05}. Also, customers knowledge is used in the process of requirements prioritization\cite{TAS06}. From the customer point of view, it is important to exercise as thoroughly as possible those functionalities that carry higher value to the users. From the developer point of view, it is important to exercise as early as possible those functionalities based on the complexity as well as inter dependencies between them. So, prioritization of test scenarios
involves both customers and developers viewpoint about the system under development. Customer prioritization is obtained directly by requiring customers to order requirements(functionality) and scenarios by importance. Developer’s viewpoint on priority is based on the knowledge of technical complexity involved both in design and coding. This knowledge is possessed by managers involved in the development process and is used for prioritization. Both these methods rely on knowledge possessed by the stakeholders[PST08].

While customer and developer inputs on priority is an influential factor in the success of a project, it is tedious especially in case of large software systems where the number of requirements and scenarios is large. Hence, there is need to aid the process of prioritization by providing automated techniques. Various work in literature use different techniques for prioritization based on factors like requirement, risk, cost and history. Metrics are used by Moisiadis[Moi00] as a measure to prioritize scenarios belonging to a use case. Actors used in each scenario, objects used in a particular scenario, level of extensions needed to exhaust the alternatives for each scenario and the number of abstract use cases used in each scenario are used as indicators to determine the importance of each scenario belonging to a use case. The advantage of the technique is that it provides an automated measure for prioritization. However, the metrics considered is limited to the use case diagram. Also, the automated technique by itself may not be effective as a sole measure to base prioritization. In this direction, this work introduces techniques that use primitives of both use case and activity diagrams as a measure of complexity to aid prioritization in addition to customer based prioritization. Metrics captured based on primitives is used in automating the process of prioritization. Also, a weighted average of customer priority and priority obtained from technical aspects(primitives of UML diagrams) provides an effective ordering of scenarios for testing.
Section 4.3.2 discusses the need for prioritization. Related work in the area is discussed in Section 4.3.3. The approach followed in this work for prioritizing use cases and scenarios is discussed in Sectin 4.3.4. Customer based prioritizatio is discussed in Section 4.3.5. Prioritization of use cases is discussed in Section 4.3.6 and prioritization of scenarios is discussed in Section 4.3.8.

4.3.2 Need for prioritization

The reasons for prioritization are:

- Prioritizing requirements and scenarios helps in understanding requirements that are important to the customer.

- In case of projects that are constrained by budget, prioritization helps decide requirements that should be considered for implementation and those that should be dropped.

- With projects that are constrained by time, prioritization helps arrange requirements in decreasing order or importance. i.e. test cases with higher priority, according to some criterion are executed earlier than those with lower priority.

- Prioritization helps build projects in iteration by assigning requirements to each iteration. This gives the customer a working product at every delivery.

- Customer conviction and satisfaction in the product increases with important requirements delivered early on.

- Prioritizing requirements help define costs and benefits for both customer and developer as well as define resource allocation.

- Prioritization helps developers concentrate time and effort on requirements of high priority.
4.3.3 Related Work on Prioritization

Test scenarios are developed from: one, the requirements independently, two, building models for testing purposes, and three, using the same analysis and design model used for development. In case of the first method, disadvantage is the effort and time required in developing the scenarios. Though requirements specification and test scenario generation are done in parallel, requirements may be interpreted differently causing mismatch. The advantage of the technique is that missing requirements may be identified during the process of developing test scenarios. In case of the second method, separate models (different from design models) are used to represent the system with focus being on testing. Modelling language used maybe the same as ones capturing requirements, like UML or different like state charts or Labelled Transition Systems(LTS). Advantage of the method is that inconsistencies and misinterpretations of requirements may be detected better due to different people building models for development and testing. The disadvantage in this case is the effort involved in developing two different models. The third approach involves using the same model to capture requirements and for testing. UML used for capturing requirements of the system are used for testing too. The use of the same model for design and testing helps in reducing effort involved in design as well as in removing inconsistencies in design. Disadvantage lies in the fact that missing and misinterpreted requirements may creep into the design and remain unidentified as the same model is used for building test cases. Automated tools for generating scenarios from UML diagrams are available(e.g. Rational, Visual Paradigm) but they do not aid in prioritizing scenarios. Addition of techniques for prioritization makes the process of testing effective.

Moisidias [Moi00] use relation between use case to prioritize scenarios. Based on UML 1.1, three relations namely communicate, extend and uses are employed to
compute priority based on object usage and actor usage. Kundu and Samanta [KS07] employ use case scenarios for prioritization. Use case scenarios are represented as system sequence diagrams and converted into a graph. Weights are assigned to edges of the graph based on the outgoing edges. The technique is automated except for the additional overhead involved in converting use case scenario into a system sequence diagram. Stallbaum et al [SMP08] introduce risk based prioritization using UML activity diagrams. Risk is calculated by using two factors, damage and probability of fault obtained by augmenting risk information for each activity during risk assessment. Sum of risk values of activities is the risk of the test scenario. Though risk information is augmented once during risk assessment, there is need to do it for all activities across requirements which is an overhead especially in case of large systems. In this work, techniques for prioritizing requirements and scenarios based on primitives of UML diagram (here, use case and activity diagrams) are used to aid in prioritization. Prioritization values thus derived do not require manual intervention can be used along with customer inputs to aid prioritization process.

4.3.4 The Approach

As mentioned previously, requirements are captured using use case diagrams. Each use case represents a functionality of the system. Activity diagrams elaborate on the functionality represented in the use case diagram. For this, a guideline was introduced: Each use case in a use case diagram is elaborated using an activity diagram. Test scenarios are generated for each use case (functionality) using the corresponding activity diagram. The use case diagrams and corresponding scenarios form a hierarchy i.e., a system has one or more actors, where each actor is associated to at least one use case. A use case is related to another by means of the <include>, <extend> and <generalization> relationship. Also, each use case
being elaborated by an activity diagram has a set of scenarios related to it.

In this section, the proposed approach to prioritize scenarios for testing is discussed. The approach consists of the following steps, with the order of execution shown in Figure 4.5.

![Figure 4.5: Steps in prioritization](image)

a. Obtain customer prioritization of use cases
b. Compute use case priority from use case diagram primitives

c. Calculate combined priority of use case

d. Compute scenario priority from activity diagram primitives

e. Calculate combined priority of scenarios

Each of the activities in the activity diagram are discussed in the following subsections in detail.

4.3.5 Customer Prioritization

Customers are one of the most important stakeholders of a software project. One of the main goals of software development is to deliver a product that is of quality and within set time schedules and cost. In this direction, there is need to understand which requirements are important to the customer and their relative order. This section discusses the need to obtain customer inputs for prioritizing requirements to aid the testing process.

The Standish Group and the British Computer Society through their study concluded that, only one in eight projects can be considered truly successful. 52.7% of completed projects cost over their original estimates whereas 1% of IT projects get cancelled before completion. Table 4.5 based on the Standish Group study[Gro] also shows that customer involvement or lack thereof contributes in a big way to the success or failure of a project.

Customers assign a value to features based on their knowledge about requirements, relative importance and the need i.e. customers prioritize requirements based upon the value that a set of features will bring to the business. Moisiadis[Moi00] in their work state that approximately 45% of the software functions are never used, 19% are rarely used, and only 36% are sometimes or always
<table>
<thead>
<tr>
<th>Rank</th>
<th>Factors for Successful Projects</th>
<th>Factors for Challenged Projects</th>
<th>Factors for Impaired Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>User involvement</td>
<td>Lack of user input</td>
<td>Incomplete requirements</td>
</tr>
<tr>
<td>2</td>
<td>Executive management support</td>
<td>Incomplete requirements</td>
<td>Lack of user involvement</td>
</tr>
<tr>
<td>3</td>
<td>Clear statement of requirements</td>
<td>Changing requirements &amp; specifications</td>
<td>Lack of resources</td>
</tr>
<tr>
<td>4</td>
<td>Proper Planning</td>
<td>Lack of executive support</td>
<td>Unrealistic expectations</td>
</tr>
<tr>
<td>5</td>
<td>Realistic expectations</td>
<td>Technology incompetence</td>
<td>Lack of executive support</td>
</tr>
<tr>
<td>6</td>
<td>Smaller project milestones</td>
<td>Lack of resources</td>
<td>Changing requirements specifications</td>
</tr>
<tr>
<td>7</td>
<td>Competent staff</td>
<td>Unrealistic expectations</td>
<td>Lack of planning</td>
</tr>
<tr>
<td>8</td>
<td>Ownership</td>
<td>Unclear objectives</td>
<td>Didn’t need it any longer</td>
</tr>
<tr>
<td>9</td>
<td>Clear vision &amp; objectives</td>
<td>Unrealistic time frames</td>
<td>Lack of IT management</td>
</tr>
<tr>
<td>10</td>
<td>Hard-working focused team</td>
<td>New technology</td>
<td>Technology illiteracy</td>
</tr>
</tbody>
</table>
used. Hence, it can be inferred that only a subset of functionalities are used mostly by the customer. Therefore, there is need to identify the subset of requirements most important to the customer. Besides, customers view of software to be built differs from developers view. To overcome this disparity, there is need to involve the stakeholders in ordering requirements.

The objective of involving the customer in prioritizing requirements are:

- Provides better understanding of requirements and clears ambiguity e.g. incomplete requirements.
- Understand customer viewpoint of requirements priority.
- Customers gain insight on the cost and technical difficulty associated with specific requirements.
- Understanding effects of change.
- Understand interrelationships among different requirements and their alignment with business requirements.
- Increased communication among stakeholders.
- Gain agreement of stakeholders.
- Increase business value to customer by identifying and testing completely the set of requirements of highest value to customer.
- Development being iterative, customer is made aware of the progress of the product.
- Implementing those requirements that are most valuable to the customer, first.
Customer-assigned priority (CP) is a measure of the importance of a requirement to the customer. Weigers [Wie99] defines two requirements prioritization scales, both subjective in nature. Requirements are grouped into three priority categories (three-level scales). In the first, three measures (high, medium and low) are used to rate requirements according to their relative importance. Another way is to prioritize requirements according to granularity. In medium to large projects, the number of functional requirements is large. Hence, there is need to choose an appropriate level of abstraction for prioritization (use case level, feature level, or details functional requirement level). Table 4.6 shows the two three-level scales.

<table>
<thead>
<tr>
<th>Names</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>critical requirement, needed for the next phase</td>
</tr>
<tr>
<td>Medium</td>
<td>required, but can be part of a later release if necessary</td>
</tr>
<tr>
<td>Low</td>
<td>functional or quality enhancement; added if resources permit</td>
</tr>
<tr>
<td>Essential</td>
<td>product not acceptable unless requirements are satisfied</td>
</tr>
<tr>
<td>Conditional</td>
<td>enhances product, but product not unacceptable if requirement absent</td>
</tr>
<tr>
<td>Optional</td>
<td>functions that may or may not be worthwhile</td>
</tr>
</tbody>
</table>

Analytic Hierarchy Process (AHP) is a technique used to obtain customer priority for requirements and scenarios [YHTS09]. It helps determine the relative merit of a requirement/scenario with reference to the set of requirements/scenarios. Asking the customer alone to prioritize requirements for development is not feasible as they prioritize requirements from their own perspective i.e. how important a requirement is to the business. Cost, technical risk, resources, interdependencies and other trade-offs associated with requirements may need that the ordering of requirements for development is different. Also, for software development, there maybe a need to follow an order i.e. a requirement may need to be built before another, due to its impact on the product architecture. Hence, there is need to involve the user and the development team in the process of prioritization.
In this work, a five point scale to prioritize requirements (very low, low, average, high, very high) is used. Customer rates a requirement on a scale of 1 to 10, 1 being a requirement which has very low priority and 10, a requirement with the highest priority. The priority assigned to requirements (use cases) by the customer is used along with priority calculated based on structural aspects of a use case diagram, which is the scope of the next section.

4.3.6 Prioritizing use cases

Use case diagrams capture the requirements of a software system. Besides representing requirements (as use cases) and the actors related to a requirement, they also capture the dependencies between requirements. In this section, data captured from the primitives of the use case diagrams are used to aid in prioritization.

4.3.6.1 Constructs and relations in use case diagrams

Constructs used in a use case diagram include: actor, use case, and the relationships between them, namely, association, include, extend and generalization as shown in Figure 4.6. An actor is a user of a system who communicates directly with the system but is not part of the system [BR07]. A use case is a functionality that a system can provide by interacting with actors. A use case may involve one or more actors. Also, a use case represents all relevant behaviour of the functionality, including normal mainline behaviour, variations on normal behaviour, exception conditions, error conditions and cancellation of a request [BR07].

The relationship between constructs are:

\footnote{A part of the work state here is published in the Proceedings of the Third UKSim European Symposium on Computer Modeling and Simulation, 2009, IEEE Xplore Digital Society}
Association: The association relationship is used between an actor and a use case. Every use case should have at least one actor, and every actor should participate in at least one use case. Also, a use case may be used by many actors and an actor can participate in many use cases.

Includes: An include relationship is used between two use cases. It incorporates once use case within the behaviour of another use case. The included use case may or may not be usable on its own.

Extends: An extend relationship is used between two use cases. The extended use case adds itself to the base use case. Also, the extended use case cannot appear alone i.e. it is not usable on its own.

Inherits: The generalization relationship is used between actors(actor - actor) and use cases(use case - use case).

An example of a use case diagram is shown in Figure 4.7.

4.3.6.2 Preprocessing

Functionality of a system can be captured using one or more use case diagrams. Use case diagrams built using a CASE tool are stored as .xmi files. Required elements from the use case diagrams of the System Under Test(SUT) is captured using an EXtensible Stylesheet Language (XSLT) processor to transform XMI
documents. Figure 4.8 shows the process involved. The data thus captured, is stored in an .xml file, a sample schema of which is given in Listing 4.1.

4.3.6.3 Computing priority from use case diagram (SP - Structural Priority)

The objective of computing priority using the constructs of the use case diagram is to concentrate effort on those functionalities that are most likely to be error prone due to structural complexity. The factors taken into consideration for prioritization of use cases include:\footnote{Metrics adapted from The Software Design Metrics tool for the UML. Available at http://www.sdmetrics.com}:

- Number of actors use case interacts with ($N_a$)
- Number of times use case appears in model ($N_t$)
Listing 4.1: Schema of use case document

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Actor" type="xs:string" maxOccures="unbounded"/>
        <xs:element name="Usecase" type="xs:string" maxOccurs="unbounded"/>
        <xs:sequence minOccures="1">
          <xs:element name="Relation" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</xs:schema>
```

- Number of use cases this use case includes ($N_{in}$)

- Number of use cases that includes this use case ($N_{ind}$)

- Number of use cases this use case extends ($N_{ex}$)

- Number of use cases that extends this use case ($N_{exd}$)

- Number of use cases inherited by this use case ($N_{ind}$)

The relation between actor and use case are of highest value as they are the points where a user interacts with the system. Hence, they get highest weight. The number of times a use case appears in a model is indicative of the importance of the functionality in the system. Hence, the factor related to appearance of a use case in the model gets next highest priority. Dependency relations are the most common relations. An include relationship denotes the inclusion of behaviour by
another use case. It also indicates potential reuse. An extend dependency indicates a relation where an extending use case continues the behaviour of the base use case. The extending use case accomplishes this by inserting additional sequences into the base use case. This is followed by the inheritance relationship also indicating potential reuse. Hence, \( N_a > N_t > N_{in} > N_{ex} > N_{ind} > N_{exd} > N_{ind} \). A weighted sum of the above factors is taken to calculate the priority of each use case. That is, \( w_1(N_a) > w_2(N_t) > w_3(N_{in}) > w_4(N_{ex}) > w_5(N_{ind}) > w_6(N_{exd}) > w_7(N_{ind}) \). \( \Sigma w_i = 1, i = 1..7 \). The use case with highest value gets highest priority.

### 4.3.7 Combined prioritization of use cases

Prioritization of use cases involves taking into consideration customer requirement as well as structural complexity. The block diagram (Figure 4.9) shows the steps involved in use case prioritization. Priority of a use case \( UC_i \) is computed as a weighted sum of customer priority and structural priority.

![Figure 4.9: Calculating priority of use cases](image)

\[
\text{Priority}(UC_i) = w_1(CP_i) + w_2(SP_i)
\]

where \( UC_i \) is the use case i,

CP is Customer Priority of \( UC_i \),
SP is Structural Priority of $UC_i$.

$w_1$ and $w_2$ are weights $| w_1 + w_2 = 1$.

Prioritization of use cases orders requirements according to their relative priority. The next step involves prioritizing scenarios belonging to each use case.

### 4.3.8 Prioritization of scenarios

The capacity of any stakeholder to prioritize scenarios is limited to test suites of small size. It is impossible to prioritize scenarios consistently in case of large test suites (i.e. test suites consisting of a large number of test scenarios). To address this problem, there is need to introduce automated techniques for prioritizing scenarios.

In this work, a technique for prioritizing scenarios based on the constructs of an activity diagram is introduced\(^6\). Each use case is elaborated using one or more activity diagrams. An activity diagram is converted into a tree and weights are given for the nodes and edges of the tree based on criteria discussed in Section 4.3.8.3. The weights are then used to prioritize scenarios. The weight associated to each type of primitive used in defining a scenario are considered to compute the weight of a path. The paths in the activity diagram are prioritized based on path-weights.

#### 4.3.8.1 The Approach

Each use case can be represented using one or more activity diagrams. Consider the example given in Figure 4.10. Activity diagrams represent the scenarios related to a use case. A scenario is a complete "path" in an the activity diagram. Execution of a set of paths in an activity diagram achieves a system function-

\(^6\)A part of the work stated here is published in the Proceedings of the 1st International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN 2009) and appears in IEEE Computer
ality. The main scenario (basic path) is the one beginning from the start node, traversing through all the intermediate nodes without any error, upto the end node. Alternate scenarios (alternate paths) are the cases when there is wrong entry of input or a condition is not satisfied that may happen during execution. An exception scenario is a path where an exception arises signalling an obstacle to executing the scenario. Based on the same, in this work, main, alternate and exception scenarios are defined as follows:

- **Main Scenario**: A main scenario is the path towards achieving a goal without any deviation.

- **Alternate Scenario**: An alternate scenario is one where one or more actions different from the main scenario is taken to achieve the goal. For example, if a node in a path has been visited more than once, for example, due to a wrong entry, then that path is an alternate scenario for the functionality represented by the use case. The post condition of the alternate scenario is the same as the main scenario.

- **Exception Scenario**: An exception scenario is one where there is an unexpected event. The post-condition is different from the post condition of the main scenario.

The objective of prioritizing scenarios extracted from the activity diagram is to identify the relative importance of scenarios. If done early in the development life cycle, it is possible to concentrate effort on designing, coding and testing the software thereby increasing quality of the software developed. Also, it ensures that customer requirements are met. The steps are:

1. Extraction of scenarios from an activity diagram.
2. Prioritizing scenarios
   a. Assign weights to nodes in paths
b. Assign weights to edges in paths

c. Calculate weight of path(scenario)

d. Normalize priority values

d. Categorize and Prioritize scenarios

Figure 4.10: Activity Diagram for use case 'Validate User'

4.3.8.2 Converting Activity Diagram into Tree Structure

The first step involves extraction of paths. Path extraction is possible by graph traversal that returns a traversal tree spanning over the graph defined in an activity diagram. An activity diagram is converted into a tree using the following steps:

1. **Activity Diagram - Graph.** Each activity corresponds to a node $n_i \in N$ of tree, $T$. The label of $n_i$ is the name of the activity. The type of the activity, namely, sequence, fork, join, branch and merge is also stored for each activity. To explain the case, activity diagram related to validating a user(refer Figure 4.10) is taken as an example. The equivalent graph for the activity diagram is shown in Figure 4.11.
2. **Traversal Tree.** Depth First Traversal is performed on the graph G to explore paths in G. Three cases are considered in traversing an activity diagram, namely, branches, loops and concurrent activities.

   **Branch.** A path starting from the start activity to the final activity constitutes a scenario. The methodology to traverse an activity diagram in case of a branch node is shown in Figure 4.12. For ease of use, the nodes in the graph and tree, have been named alphabetically.

   **Loop.** When traversing an activity diagram, a restriction is imposed that loops be executed at most twice to avoid the problem of path explosion. A modified depth first search (DFS) algorithm is used for this. Hence, an activity may be visited for a maximum of two times [XLL07, MXX06, KKBK07].

   The above constraint is used to generate the tree shown in Figure 4.13(c) from the activity diagram of Figure 4.13(a). For ease of use, the nodes in the tree, T have been named alphabetically. It can be seen from the tree that activities b, c, d and e occur for a maximum of two times as they form part of a loop.

   **Concurrent Activities.** Concurrent activities are those that are executed simultaneously (i.e. in parallel). The two horizontal splits indicate the start and
Figure 4.12: Conversion to Tree - Branch

Figure 4.13: Conversion to Tree - Loop
end of concurrent activities. In Figure 4.14(a), Validate_Pin() and Validate_Card() are concurrent activities which can execute in any order. Both the activities must be completed at the join node for further execution. The above constraint is used to generate the tree shown in Figure 4.14.

![Activity diagram for 'Access card Validation'](image)

![Graph representation](image)

![Corresponding tree](image)

Figure 4.14: Conversion to Tree - Concurrent Activities

4.3.8.3 Prioritizing scenarios

The tree thus obtained from an activity diagram can be used for prioritizing scenarios. It may be noted that each path from the start node to the end node (root node to leaf node in tree T), corresponds to a scenario of the use case. The steps given below are used to prioritize the scenarios.

1. **Assign weights to nodes in T**

The different constructs used in an activity diagram are action/activity, branch, merge, fork and join. Weights are assigned to each of these nodes based on the complexity and possibility of occurrence of defects. The fork-join construct rep-
resents concurrent activities. The branch-merge nodes represent decisions. The complexity values are given keeping in mind the risk factor involved with reference to the primitives. The fork-join nodes handle concurrency and are hence given highest weight. The branch-merge nodes need to be tested to check the boolean expression for all possible branches to be followed. Hence, second highest weightage is assigned to branch-merge nodes. Action/activity nodes are given a weightage of 1.

\[
Wt(n) = \begin{cases} 
3 & \text{for fork-join nodes} \\
2 & \text{for branch-merge nodes} \\
1 & \text{for action/activity nodes} \\
.5 & \text{for activities inside fork-join} \\
0 & \text{for start and stop nodes}
\end{cases}
\]

Figure 4.15: Tree T after assigning node weights

The result of traversing the tree using depth-first algorithm and assigning weights to the nodes in 4.12(a) based on the above equation is shown in Figure 4.15.
2. Assign weights to edges in \( T \)

\[
Wt(e) = (n_{i})_{in} \times (n_{j})_{out}
\]

where \((n_{i})_{in}\) is the number of incoming dependencies of node \( n_{i} \) and \((n_{j})_{out}\) is the number of outgoing dependencies of node \( n_{j} \) and \( e \) is the edge connecting \( n_{i} \) and \( n_{j} \). The tree, \( T \), after assigning weights to the edges based on the above equation is shown in Figure 4.16. Again, depth first traversal was used to assign weights to edges in tree, \( T \).

![Tree T after assigning edge weights](image)

**Figure 4.16: Tree T after assigning edge weights**

3. Calculate weight of path

The equation given below is used to calculate the weight of the paths. The sum of the weights of both the nodes and edges is the weight of the path.

\[
Wt(\text{path}) = \sum_{i=1}^{n} Wt(N_{i}) + \sum_{j=1}^{m} Wt(E_{j})
\]

Table 4.7 shows the weights of the five paths. The weight of the path is then used for prioritization of scenarios.
Table 4.7: Calculated weights for the five paths

<table>
<thead>
<tr>
<th>Path</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A→B→C→D→E→B→C→D→E</td>
<td>24</td>
</tr>
<tr>
<td>A→B→C→D→E→B→C→D→F</td>
<td>24</td>
</tr>
<tr>
<td>A→B→C→D→F</td>
<td>13</td>
</tr>
<tr>
<td>A→B→C→A→B→C→D→F</td>
<td>20</td>
</tr>
<tr>
<td>A→B→C→A→B→C→D→E</td>
<td>20</td>
</tr>
</tbody>
</table>

4. Normalize priority values
As mentioned before, a scale to prioritize scenarios (very low, low, average, high, very high) is used. To convert priority values to a scale of 1 to 10, normalisation is done on priority values obtained from a set of scenarios belonging to an activity diagram.

5. Categorize scenarios
Scenarios are categorized by the actions [JCSdPLK00]. Scenarios are of three types: main scenarios, alternate scenarios and exception scenarios as discussed previously in Section 4.3.8.1.

4.3.8.4 Calculating priority of scenarios
The priority of a scenario is the combined priority of a scenario plus the priority of the corresponding use case. Thus, priority of scenario $s_i$ related to use case $uc_j$ is given as:

$$\text{Priority}(s_i) = w_1(\text{Priority}(uc_j)) + w_2(\text{Priority}(s_i))$$

$w_1$ and $w_2$ are weights $| w_1 + w_2 = 1$.

This combined priority is used for ordering scenarios by type. Main scenarios are considered first, followed by exception and alternate scenarios. Figure 4.17 shows the prioritized scenarios of the 'Validate User' activity diagram. The order of prioritization $P_1, P_2, P_3, P_4, P_5$ is based on the definitions stated above. Table 4.8
shows prioritized scenarios categorized based on the type of scenario. Highest priority is given to main scenarios and then to alternate and exception scenarios. The path with highest weight gets highest priority.

### 4.3.9 Prioritization: Summary

In summary, this section presented techniques for prioritizing requirements (use cases) from use case diagrams and scenarios from activity diagrams based on the primitives defined by OMG. Prioritization values thus obtained are used along with prioritization values obtained from customers. A weighted sum of the two is taken to compute priority of use case or scenario. The advantages of the methodology are:
• The technique considers both customers as well as developers viewpoint for prioritization of scenarios.

• Customer knowledge on priority may differ. To incorporate this, weights are given for customer priority as well as technical priority. Weights can be adjusted based on customer knowledge of requirement priority.

• Priority calculated using primitives of use case and activity diagrams is automated and does not require manual intervention.

• This technique may be combined with other techniques to aid prioritization.

In this work, prioritization of scenarios by the customer is not considered. However, in case it is available, a weighted sum similar to the method used for use cases can be applied. Based on priority, scenarios are ordered/ranked for further consideration particularly in case of integration and system testing. As the number of scenarios even for a medium size application could be large, there is a need for selecting scenarios in order of their importance based on some defined context. In the next section, scenario selection is discussed.
4.4 Scenario Selection

4.4.1 Introduction

Specification based testing involves generating test cases from specification, here, UML. The number of automatically generated test scenarios from UML activity diagrams, is large. A test suite is composed of a set of test cases wherein each test case covers a set of actions to be performed by the user(actor) of the system. Given limited resources in comparison to the size and complexity of software, there is need for early and effective feedback with reference to defects in software.

As mentioned earlier, functional requirements are recorded using use case diagrams. Each use case is elaborated using activity diagrams i.e. a use case may have one or more activity diagrams representing main, alternate and exception flows. Automated scenario generation from activity diagrams is done based on methodology followed in Section 4.2. It is considered that loops in activity diagrams are traversed twice. Thus, for each use case, a set of scenarios is obtained whose size can be large dependent on the size and complexity of activity diagrams.

A look at scenarios thus generated reveals that parts of test scenarios are common. A study of similarity among a given set of scenarios is carried out for reducing effort in testing the scenarios and concentrating on scenarios that are different or dissimilar. The objective of the approach is to reduce effort involved in running scenarios that are highly similar by picking a representative of such scenarios and including more dissimilar scenarios with the aim of increasing fault detection and coverage. Test scenario selection thus involves selecting a subset of scenarios that best represent the set of scenarios. A threshold value ’t’ (percentage of test scenarios to be selected) is obtained from the quality assurance team engineers. It is essential to select scenarios within the threshold, such that the selection ensures qualitative as well as quantitative success in testing a system. In
this section, techniques for automated test selection are proposed.

An automated strategy for test scenario selection based on the use of Levenshtein distance [Lev65] as a measure of dissimilarity between scenarios is presented in Section 4.4.4. For each pair of scenarios, the Levenshtein distance is calculated. The least value indicates scenario pairs that are least dissimilar and vice versa. Hence, selection is based on the highest dissimilarity between scenarios. A second technique for test case selection presented is based on the Longest Common Subsequence applied to subscenarios(Section 4.4.5). A subscenario is a contiguous chain of activities within a scenario. The technique looks at similarity between scenarios in terms of the length, weight and position of the common subscenarios in the scenario. The above factors are used to calculate the similarity between each possible pair of scenarios. A similarity value indicates the degree of structural similarity i.e. less the value, more similar they are. Again, a heuristic is used to pick one of the two scenarios. Clustering is the grouping of objects(here scenarios) of similar kind into categories. In this work, hierarchical as well as partitional clustering algorithms are used to cluster scenarios according to a distance measure(Levenshtein distance), such that elements in the same cluster are similar based on some criteria(Section 4.4.6).

Generally, selection of one scenario between two is done randomly [CNM07, YH07]. In this work, heuristics other than random choice are introduced to capture the added advantage of using information available on test scenarios i.e. priority, type, knowledge of the quality assurance team (preferred selection) in the selection process thereby producing better results. The test suite thus obtained is used for testing either directly(for random testing) or forms input for prioritization.

Related work in the area of test case selection is discussed in Section 4.4.2. Section 4.4.3 discusses criteria used to measure the effectiveness of selection techniques. Section 4.4.4 presents a technique for clustering based on Levenshtein
distance. Section 4.4.5 presents a similarity measure based on the length and position of a subscenario in a scenario. Selection technique based on Agglomerative Hierarchical Clustering is described in Section 4.4.6.

### 4.4.2 Related Work on Test Case Selection

Test case selection provides several benefits in automated testing process. Exhaustive testing being impossible, there is need to determine a subset of test cases that ensure test objectives, namely, maximum coverage, and early fault detection. For this, there is need to select an effective subset of the original test suite. Several techniques have been developed which attempt to maintain quality of throughput by reducing the size of the test suite and at the same time increasing efficiency.

Similarity measures for test case selection have been used by [CANM08, CNM07, CLM04]. Test cases are selected based on the distance value computed between two test cases. Techniques used include pairwise comparison in [CNM07] and Euclidean distance in [CLM04]. Cartaxo et al. use Labeled Transition Systems (LTSs) as the model from which they obtain test cases. In [CLOM06], the authors extend the idea of distance to testing object oriented programs by defining a distance for objects, the 'object distance'. The object distance computes distances between arbitrary objects. They describe a model for representing the differences between two objects, and use Levenshtein distance to calculate distance between class names, object names and its contents.

Burguillo et al [BLFR02] in their work consider heuristic driven techniques for test selection. They use four factors, namely, risk, coverage, cost and efficiency, which helps in classification and selection of test cases according to different criteria. Risk based test selection is the focus of work done by Chen et al [CP03b]. Risk is based on the cost of each test case(valued by both customer and vendor) as well as severity. Risk exposure(RE), a product of cost and severity is taken
as the basis for test selection. Scenarios that cover most critical cases are considered first. Cow Suite tool by [BBM02] use UML use case and sequence diagrams to record requirements. Each use case diagram is elaborated using a sequence diagram, to scenarios forming a tree structure. Weights are assigned based on functional importance such that the sum of weights at any level equals to one. The test generation algorithm used by Cow Suite generates all possible test cases. Selection is done based on the weight of the scenarios obtained by product of weights of all nodes in the path leading to particular scenario.

Activity diagrams are used for regression test selection in [CPS02]. Regression test selection techniques involves selecting a subset of test cases to determine if the modified program has the same behaviour as a previous, acceptable version of the program running on T, a set of test cases. A framework for analyzing regression test selection techniques is presented in [RH96]. The framework consists of four categories: inclusiveness, precision, efficiency, and generality. They analyze different techniques on the four factors. In [RH97], the authors construct control flow graphs (CFG) for a procedure or program and its modified version. They use the CFGs to select tests that execute changed code from the original test suite.

In this work, use case diagrams model functionality of a system. Each use case is elaborated using activity diagrams. Scenarios are generated from activity diagrams which form the input of the test selection process. This work also uses distance measure to determine similar scenarios, like work in [CANM08, CNM07, CLM04]. However, in this work different measures are used to select between two scenarios. Information like scenario type (main, alternate, exception) and priority are available as inputs provided by the quality assurance team/customers or through automated techniques presented in previous sections. This information can be used to help in better selection of scenarios.
4.4.3 Selection Criterion

Similarity based test selection is based on distance measure or a similarity measure that gives a value indicating similarity between two scenarios i.e. given two scenarios with a high similarity value, there is need to pick one of the two. Current work in the area use random selection to pick scenarios by selecting one or the other. To improve the criteria for picking one of the two scenarios, other heuristics can be considered based on properties of the scenarios like priority and type. One or a combination of heuristics can be used for picking one of the scenarios. The heuristics used in this work is given below:

**Random** : One of the two scenarios, is selected randomly.

**Priority based** : Scenario with higher priority is selected. Priority of scenario is calculated based on one or a combination of the techniques: customer assigned, statement coverage, risk based, random, etc. [SWO05, RUCH01, EMR02]. In case of equal priority, a scenario is selected randomly.

**Subsumption** : Given two test scenarios, X and Y, X subsumes Y if and only if the scenario Y is contained in X in that order. Hence, X is selected.

**Type** : Main scenario(flow) gets priority over alternate and exception scenario.

**Preferred Scenarios** : Preferred scenarios are the subset of all scenarios listed by the quality assurance engineers as those that must compulsorily be tested. All such scenarios are included in the test suite followed by selection using one of the above techniques.
4.4.4 Scenario Selection based on Levenshtein Distance

4.4.4.1 Introduction

Levenshtein distance [Lev65], named after Vladimir Levenshtein, is a metric for measuring the amount of difference between two sequences (i.e., the so called edit distance). The Levenshtein distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. A modification of the Levenshtein distance calculates genetic distance. Genetic distance between two words is taken as the edit distance divided by the number of characters of the longer of the two. The genetic distance is thus, any value between 0 and 1. The algorithm has a complexity of $\Theta(mn)$, where $m$ and $n$ are the lengths of the strings.

For example, consider two strings,

T E S T I N G and

T E S T E D

Edit distance : 3 (Substitute ‘I’ by ‘E’; substitute ‘N’ by ‘D’; and delete ‘G’)

Genetic distance : 0.43 (Edit distance 3 divided by 7, length of longer string)

The algorithm to calculate the Levenshtein distance between two strings is given in Algorithm leven(s1,s2).

4.4.4.2 Application to Test Case Selection

For each use case, $uc_i$ a set of scenarios, $S$, deduced from all activity diagrams elaborating the use case is obtained. Each of these scenarios is specified as a string of characters where each character represents an activity. The Levenshtein distance
Algorithm 4 leven(s1, s2) // calculates Levenshtein distance between two scenarios

1:  \{  
2:  \textbf{Input :}  
3:  \hspace{1em} s1, s2 : scenarios of size m and n.
4:  \textbf{Output :}  
5:  \hspace{1em} \text{Distance value between scenarios}
6:  \textbf{Initialize :}  
7:  \hspace{1em} d = \text{a matrix of size } m \times n, \text{ set to 0}
8:  \}

9:  \textbf{for } i = 0 \text{ to } m \text{ do}
10:  \hspace{1em} d[i,0] = i
11:  \textbf{end for}

12:  \textbf{for } j = 0 \text{ to } n \text{ do}
13:  \hspace{1em} d[0,j] = j
14:  \textbf{end for}

15:  \textbf{for } j = 1 \text{ to } n \text{ do}
16:  \hspace{1em} \textbf{for } i = 1 \text{ to } m \text{ do}
17:  \hspace{2em} \textbf{if } (s1[i] = s2[j]) \text{ then}
18:  \hspace{3em} d[i,j] = d[i-1,j-1]
19:  \hspace{2em} \textbf{else}
20:  \hspace{3em} d[i,j] := \text{minimum}(d[i-1,j] + 1, d[i, j-1] + 1, d[i-1, j-1] + 1)
21:  \hspace{2em} \textbf{end if}
22:  \hspace{1em} \textbf{end for}
23:  \textbf{end for}

24:  \textbf{if } (m > n) \text{ then}
25:  \hspace{1em} \text{return } d[m,n]/m
26:  \textbf{else}
27:  \hspace{1em} \text{return } d[m,n]/n
28:  \textbf{end if}
(genetic distance) is used to calculate the dissimilarity between two scenarios\textsuperscript{7}. The genetic distance between each pair of scenarios pertaining to a use case is stored in a matrix $M = n \times n$ where $n$ is the number of scenarios. The minimum value in the matrix belongs to the scenarios, say, $s_1$ and $s_2$ i.e. having minimum dissimilarity. Selection criterion described in Section 4.4.3 is used to select/pick one scenario between the two.

### 4.4.4.3 Algorithm

Algorithm $\text{Select}(S, P, Tq)$ shows the steps involved in selecting test cases using Levenshtein genetic distance. $S$, the set of scenarios and $P$, the percentage of test cases to be selected is given as input. $Tq$ denotes the selection technique to be adopted. $D$ is the matrix, of size $n \times n$, containing the Levenshtein distance calculated between scenarios. $T$ is the threshold calculated based on the percentage of test cases to be selected i.e. the number of test cases to be selected. First, the Levenshtein genetic distance for the scenarios is calculated. Then, selection of scenarios is done based on the technique (random, priority, type) selected. In case two scenarios are equal, i.e. two scenarios have same priority, then random selection is done.

### 4.4.4.4 Example

Consider as an example, Figure 4.18 showing the graph derived from an activity diagram. Activities are represented as nodes and transitions as edges. Using the test scenario generation algorithm, all paths in the graph are deduced. The\textsuperscript{7} A part of the work stated here is published in the Proceedings of the 6th International Conference on Distributed Computing and Internet Technologies (ICDCIT 2010) and appears in Springer-Verlag.
Algorithm 5 Select(S,P,T_q) // selection of a subset of scenarios

1: { Input:
    S : the set of scenarios.
P : percentage of test cases to be selected.
T_q : the selection technique adopted.
n : size of S, the set of scenarios
T : threshold for test case selection (n times P/100)
Output:
    Selected subset of Scenarios
Functions called:
    D(min) : returns the scenarios having minimum value in D
    SelectScenario(T_q,r,c) : returns scenario selected according to technique T_q
    levenshtein(S) : returns a matrix containing distance values between scenarios
Initialize:
    count = 0
    D = matrix of size n × n, containing Levenshtein distance between scenarios
    TS = test suite after selection
}
2: D = levenshtein(S)
3: while (D <> 0) do
4:     // while matrix D is not null
5:     if (count < T) then
6:         // if number of scenarios selected is less than threshold, T
7:          (r,c) = min(D) // r and c are the scenarios having minimum value in D
8:          result = SelectScenario(T_q,r,c) // result contains the scenario selected w.r.t T_q
9:         Add scenario 'result' to TS
10:        Remove scenario 'result' from D
11:        count = count + 1
12:     else
13:         exit
14:     end if
15: end while
Figure 4.18: Set of Scenarios (i) Graph (ii) 5 out of 9 scenarios generated
activity diagram consists of nine scenarios (paths from the start node to the end node) five of which is shown in Figure 4.18(ii). Here, loops are executed at most twice. For brevity, only activity names are considered. Table 4.9 shows the set of scenarios generated.

Table 4.9: Set of scenarios generated from graph in Figure 4.18

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.27</td>
<td>.33</td>
<td>.43</td>
<td>.5</td>
<td>.58</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>.42</td>
<td>.21</td>
<td>.57</td>
<td>.42</td>
<td>.27</td>
<td>.39</td>
<td>.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>.5</td>
<td>.25</td>
<td>.37</td>
<td>.2</td>
<td>.33</td>
<td>.45</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>.56</td>
<td>.37</td>
<td>.33</td>
<td>.22</td>
<td>.36</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.16</td>
<td>.31</td>
<td>.39</td>
<td>.27</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.21</td>
<td>.26</td>
<td>.14</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.17</td>
<td>.32</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.32</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.10: Levenshtein distance calculated between scenarios in Table 4.9

The matrix D with the pairwise Levenshtein distance (genetic distance) calculations for the scenarios listed are shown in Table 4.10. Values indicate level of dissimilarity between the scenarios. Consider that the desired path coverage percentage is 50%. Then, it is required to select four test cases out of a total set of nine scenarios. Hence, threshold value = 4.

The scenarios with minimum dissimilarity is between scenarios '[6]' and '[9]' having Levenshtein distance = 0.14. Assuming that 'Random' is the selection criterion, scenario '[9]' is selected. Scenario '[9]' is removed from the matrix. Table 4.11 shows matrix D after elimination of scenario '[9]'. The next minimum value is 0.16 between scenarios '[5]' and '[6]'. This procedure continues until the
threshold value is reached. The test suite with 50% coverage consists of scenarios [9],[6],[8] and [7].

Table 4.11: Distance table after selection of scenario 9

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.27</td>
<td>.33</td>
<td>.43</td>
<td>.5</td>
<td>.58</td>
<td>.47</td>
<td>.56</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.42</td>
<td>.21</td>
<td>.57</td>
<td>.42</td>
<td>.27</td>
<td>.39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.5</td>
<td>.25</td>
<td>.37</td>
<td>.2</td>
<td>.33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>.56</td>
<td>.37</td>
<td>.33</td>
<td>.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.16</td>
<td>.31</td>
<td>.39</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>.21</td>
<td>.26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>.17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4.5 Scenario selection based on Common Substrings

A subscenario is a contiguous chain of activities within a scenario. In this work, the idea of substrings is used to introduce a new similarity measure that considers the common subscenarios, their lengths, and weight with respect to the position in the scenario. These factors are used to calculate the similarity between each possible pair of scenarios. A similarity value indicates the degree of structural similarity i.e. more the value, more similar they are. The pairs are arranged in ascending order of their similarity value. A bottom up approach is followed in selecting pairs and selection of pairs continue until the desired test coverage criteria is met. From each pair, one path is selected as test scenario.

4.4.5.1 The Approach

The common substring between two strings is the set of contiguous set of characters in a string. The same idea is adapted for selection of test scenarios. In this work, a common subscenario is the longest contiguous chain of activities that exists in

---

8A part of the work stated here is published in the Proceedings of the 4th International Conference on Industrial and Information Systems, 2009 and appears in IEEE Xplore Digital Library
both scenarios. Common subscenario can be viewed in two terms: it can be said that longer the subscenario, better is the match between two scenarios i.e. they are similar; secondly, if the sum of the lengths of all common subscenario is close to the length of larger scenario, again, it can be said that the two scenarios are highly similar.

Based on the above, a new measure is introduced that takes into consideration common subscenario in a scenario, their length as well as relative position in the scenario. It is assumed that length of the common subscenario must be a value between 3..n considering that a subscenario is of the form $<activity><activityedge><activity>$

where an activity edge is an open arrowhead line connecting two activity nodes. For brevity, only activity names are consider here.

The intention is: one, by considering common subscenarios, similarities between scenarios is taken into account; two, length of the subscenario determines number of contiguous activities that are similar; and three, different weights can be given based on the position of the subscenario. Activity diagrams elaborate a functionality. As can be observed, activity diagrams can be sliced based on the importance or risk value of the activities. Quality assurance engineers define the slice and assign weights to the slices such that

$$W = \{w_1, w_2, ..., w_n\} \mid \Sigma w_i = 1.$$  

Figure 1 shows an activity diagram sliced into three. The heuristic used is that slice 2 containing computational aspects be given more weightage than slice 3 where results are generated, which is higher than slice 1 wherein inputs are obtained.

On basis of the above, a measure is defined that reflects length and relative importance of the subscenario to the size of the scenario.
If \( s_1 \) and \( s_2 \) are two scenarios, then,

\[ S_b = \{s_{b_1}, s_{b_2}, \ldots, s_{b_n}\} \] is the set of subscenarios, common between \( s_1 \) and \( s_2 \);

\[ W = \{w_1, w_2, \ldots, w_n\} \] is the relative weights of the subscenario in \( S_b \) with respect to position in scenario. \( w_i \) takes values between 0 and 1. If weights are different, then minimum of the weights is subtracted with the displacement value (i.e. if subscenarios are similar but at differing positions, then the weight is lesser than if the subscenarios were at the same positions in the scenarios). If both the scenarios are equal, then weight=1. If both scenarios do not have any common subscenarios, then the weight=0.

\[ \text{length}(s_{b_i}) \] returns the length of the subscenario.

\[ \text{len} = \text{length}(s_1), \text{if length}(s_1) > \text{length}(s_2); \text{else, length}(s_2). \]

Then,

\[ \text{Similarity}(s_1, s_2) = \frac{\sum(w_i \times (\text{length}(s_{b_i})))}{\text{len}} \quad (4.1) \]

For example, consider two scenarios (Figure 4.19),

\( s_1 = "A\ B\ C\ D\ E\ F\ G\ N" \) and

\( s_2 = "A\ B\ C\ H\ I\ J\ K\ N". \)

The activity diagram pertaining to scenarios are sliced into three, and assigned
weights, 0.3, 0.5 and 0.2.

Then, \( S_b = \{"A \ B \ C"\} \)

\[ W = \{0.3\} \]

\[
\text{Similarity}(s_1, s_2) = \frac{0.3 \times 3}{8} = .11
\]

The value 0.11 is the similarity value between the two scenarios.

### 4.4.5.2 Example

Consider that 50% scenarios from Figure 4.19 need to be selected. Values calculated using similarity measure is shown in Table 4.12.

Table 4.12: Distance values between scenarios using similarity measure for Figure 4.19

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.21</td>
<td>.11</td>
<td>.09</td>
<td>.07</td>
<td>.07</td>
</tr>
<tr>
<td>2</td>
<td>.09</td>
<td>.09</td>
<td>.06</td>
<td>.06</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.21</td>
<td>.12</td>
<td>.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>.10</td>
<td>.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>.20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Picking the largest value, 0.21, the highest similarity, one scenario is selected randomly between 1 and 2, say 1, and added to the test suite, TS. Then, scenario '1' is deleted from the matrix(Table 4.13), and the same procedure continues until 50% of scenarios is selected. TS = 1,3,5.

Table 4.13: Distance matrix after deletion of scenario 1

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>.09</td>
<td>.09</td>
<td>.06</td>
<td>.06</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.21</td>
<td>.12</td>
<td>.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>.10</td>
<td>.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>.20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

168
4.4.5.3 Algorithm

Algorithm Select(S, P, Tq) shows the steps involved in selecting test cases using similarity measure. SC, the set of scenarios and P, the percentage of test cases to be selected is given as input. D is the matrix, of size $n \times n$, containing the distance calculated between scenarios. T is the threshold calculated based on percentage of test cases to be selected i.e. number of test cases to be selected. First, distance between scenarios is calculated. Then, selection of scenarios is done randomly.

**Algorithm 6 Select(S, P, Tq) //calculate distance between two scenarios**

1: {  
2:     \textbf{Input :}  
3:     S : the set of scenarios  
4:     P : the percentage of test cases to be selected  
5:     n : size of S, the set of scenarios  
6:     T : Threshold for test case selection (n X P/100)  
7: \textbf{Output :}  
8:     Distance value between scenarios  
9: \textbf{Initialize :}  
10:     count = 0  
11:     D = matrix of size $n \times n$, containing similarity values between scenarios  
12: \textbf{Functions :}  
13:     max(D) is a function that returns the scenarios having maximum value in D  
14:     Selectscenario(r,c) is a function that randomly selects one of the scenarios r,c  
15:     Similarity(S) is a function that returns the distance calculated according to equation 1  
16: }  
17: D = Similarity(S)  
18: \textbf{while (D <> 0) do}  
19: \text{ // while matrix D is not null }  
20:     \textbf{if (count < T) then}  
21: \text{ // if number of scenarios selected is less than threshold, T }  
22:     r,c = max(D)  
23: \text{ // r and c are the scenarios having maximum value in D }  
24:     result = Selectscenario(Tq,r,c)  
25: \text{ Add scenario 'result' to TS }  
26:     Remove scenario 'result' from D  
27: \textbf{else}  
28: \text{ exit }  
29: \textbf{end if}  
30: \textbf{end while}  

4.4.6 Clustering based Test Selection

4.4.6.1 Introduction

Clustering is the grouping of objects (here scenarios) of similar kind i.e. it partitions objects into mutually exclusive groups. Thus, each group (cluster) consists of objects that are similar among themselves and dissimilar to objects of other groups. Cluster analysis identifies and classifies objects on the basis of the similarity of the characteristics they possess minimizing intra-group variance and maximizing inter-group variance as shown in Figure 4.20. Clustering therefore results in a number of heterogeneous groups with similar (homogeneous) content.

The objective of clustering are:

- discover types from given data
- obtain a sample of several types instead of a big sample set

![Figure 4.20: Diagrammatic representation of clustering](image)

Similarity criterion is the distance between two or more objects belonging to the same cluster. Techniques that use distance as a means for grouping is called distance based clustering.

Hamming distance is used as the basis for clustering by Yoo et al [YHTS09]. Dynamic execution trace of each test case is used to calculate similarity between features tested. The limitation is that the method requires an execution trace to
be generated from code for clustering. In this work, clustering is used to group scenarios based on similarity obtained from scenarios. Similarity between scenarios is determined in terms of activities and transitions between them using Levenshtein distance. A look at Figure 4.19 shows activities and transitions that are common between scenarios. As mentioned before, exhaustive testing being impossible due to constraints of resources and time, there is need to determine an effective set of scenarios. The commonality in activities and transitions across scenarios (similarity) is used in this work to cluster scenarios. Clustering produces groups of scenarios such that members of a group are similar. One or more scenarios from each group can be used for testing.

In this work, Agglomerative Hierarchical Clustering (AHC) is used for clustering scenarios generated from activity diagrams. Given a set of clusters, and a percentage of scenarios to be selected, scenarios are randomly picked from each cluster. The set of scenarios thus obtained forms a test suite (random ordering). Also, the set of scenarios may be ordered (prioritized) according to some criterion (priority, type) and then executed.

4.4.6.2 Selection Method

Agglomerative Hierarchical Clustering (AHC) technique is used for clustering scenarios.

**Agglomerative Hierarchical Clustering**

Agglomerative Hierarchical Clustering is a bottom-up clustering method. An AHC clustering procedure produces 'n' single scenario clusters $P_n, P_{n-1}, \ldots, P_1$. $P_n$ consists of n single object 'clusters' which in this case are individual scenarios. At each stage, two clusters which are closest (most similar) are joined to form a
new cluster. Finally, $P_1$ consists of a single group consisting of all n objects (here, scenarios).

Figure 4.21 shows the clusters formed by AHC, algorithm for which is given below. The dendrogram is a tree structure that represents clusters. Also, by cutting the tree at different heights, it is possible to generate k clusters for any k in $[1, n]$.

Algorithm 7 gives the steps involved in Agglomerative Hierarchical Clustering technique. The resulting dendrogram, D, is a tree structure. Similarity value between scenarios is calculated using a similarity/dissimilarity metric, namely, Levenshtein distance. The similarity matrix gives the set of scenarios $s_1, s_2$ that are similar which is used in the algorithm to determine the clusters.

To obtain a test suite, 'k', the level indicating number of clusters is obtained. All clusters at level 'k' with scenarios belonging to each cluster is returned. To select a percentage of scenarios from the clusters for the test suite, scenarios within a cluster are selected randomly. Again, it would be disadvantageous to run all scenarios belonging to a cluster before executing scenarios from the next as similar scenarios belonging to a cluster get executed before execution of the next cluster.
Algorithm 7 $AHC(S, n)$ // Clustering of a set of scenarios, of size n

1: \{ Input: \[S : A set of scenarios\] \[Output: A dendrogram, D, giving all clusters.\] \[Initialize: Cl : Cluster \}\}
2: Form $n$ clusters with each scenario.
3: Add the clusters to 'Cl'.
4: Calculate similarity value between scenarios using Levenshtein distance, leading to matrix, $M$.
5: while (Cardinality(Cl) > 1) do
6: Find pair of scenarios, $s_i$, $s_j$ with minimum distance from $M$.
7: Merge the pair to form a new cluster, $C_n$.
8: Remove the scenarios from $M$.
9: Remove the scenario pair from Cl.
10: Add $C_n$ to Cl.
11: Insert $C_n$ as parent of scenarios $s_1$, $s_2$ into D.
12: end while

This would lead to similar faults getting detected. To avoid such a case, scenarios are picked for execution in one of the following three ways:

- The set of scenarios obtained from the clusters after selection are ordered again, according to some criterion.

- One scenario from each cluster is picked for execution in order till scenarios get exhausted.

- Random selection of set of scenarios obtained from the clusters.

4.4.6.3 Example

Consider the distance matrix for the activity diagram in Figure 4.19. Values calculated using similarity metric is shown in Table 4.14.

Each scenario is taken as a cluster. In each step of the iteration, the closest pair of clusters is taken into consideration. In this case, the closest cluster is between cluster 3 and 4 with shortest distance of 0.2. Therefore, cluster 3 and 4 are grouped
Table 4.14: Distance calculated between scenarios using metric for Figure 4.19

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>.30</td>
<td>.63</td>
<td>.70</td>
<td>.71</td>
<td>.75</td>
</tr>
<tr>
<td>2</td>
<td>.30</td>
<td>0</td>
<td>.60</td>
<td>.60</td>
<td>.70</td>
<td>.70</td>
</tr>
<tr>
<td>3</td>
<td>.63</td>
<td>.60</td>
<td>0</td>
<td>.20</td>
<td>.50</td>
<td>.50</td>
</tr>
<tr>
<td>4</td>
<td>.70</td>
<td>.60</td>
<td>.20</td>
<td>0</td>
<td>.60</td>
<td>.60</td>
</tr>
<tr>
<td>5</td>
<td>.71</td>
<td>.7</td>
<td>.5</td>
<td>.6</td>
<td>0</td>
<td>.25</td>
</tr>
<tr>
<td>6</td>
<td>.75</td>
<td>.7</td>
<td>.5</td>
<td>.6</td>
<td>.25</td>
<td>0</td>
</tr>
</tbody>
</table>

into cluster (3, 4). Then the distance matrix is updated (see Table 4.15). Distance between ungrouped clusters do not change in the distance matrix. To calculate the distance between newly formed clusters and other clusters, single linkage rule is used, where the minimum distance between original cluster and the combined clusters are taken.

Table 4.15: Distance matrix after clustering scenarios 3/4

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3/4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>.30</td>
<td>.63</td>
<td>.71</td>
<td>.75</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.30</td>
<td>0</td>
<td>.60</td>
<td>.70</td>
<td>.70</td>
<td></td>
</tr>
<tr>
<td>3/4</td>
<td>.63</td>
<td>.60</td>
<td>0</td>
<td>.50</td>
<td>.50</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.71</td>
<td>.7</td>
<td>.5</td>
<td>.6</td>
<td>0</td>
<td>.25</td>
</tr>
<tr>
<td>6</td>
<td>.75</td>
<td>.7</td>
<td>.5</td>
<td>.6</td>
<td>.25</td>
<td>0</td>
</tr>
</tbody>
</table>

The process continues until there is only one cluster. Dendrogram, D, for the above is given in Figure 4.22. Given a set of clusters, and a percentage of
scenarios to be selected, scenarios are randomly picked from each cluster. The set of scenarios thus obtained forms a test suite (random ordering). Also, the set of scenarios may be ordered (prioritized) according to some criterion (priority, type) and then executed.

4.4.7 Scenario Selection: Summary

To summarize, in this section, techniques for test scenario selection is presented. One class of technique use distance measure, one based on Levenshtein distance and another on the Longest Common Subsequence for selection of scenarios. The second class, uses clustering based on the distance measure as a way of selecting scenarios to form a test suite.

Levenshtein distance is used to calculate the similarity between two scenarios based on the number of insertions, deletions and substitutions required to convert one scenario to another. The technique is simple. However, the technique looks at the activities and transitions individually rather than as a subscenario. A second distance measure incorporates a subscenario as being the unit of similarity and adapts the Longest Common Subsequence algorithm. A third technique based on clustering is presented. The Agglomerative Hierarchical Clustering technique is used to cluster scenarios based on a similarity measure, here distance (Levenshtein distance). Given the percentage of scenarios to be considered and the number of clusters, scenarios are selected from the clusters in random.

Thus, a set of scenarios, which is a subset of the original set maybe selected to form a test suite.
4.5 Summary

Techniques for generation, prioritization and selection of scenarios to form a test suite is presented. Automated generation of scenarios from UML use case and activity diagrams leads to large number of scenarios. Given constraints of resource and time in testing the software, there is need for automated techniques for generation, prioritization and selection of scenarios.

Concurrent activities in an activity diagram are represented using fork-join constructs. As the order of execution of concurrent activities cannot be predetermined, scenario generation explores all possibilities. However, it is impossible to test exhaustively. To overcome this, domain dependency existing between concurrent activities is used by assigning priority to activities or categorizing them into levels. The advantage of the approach is that only valid scenarios are generated thereby reducing considerably the number of scenarios generated for testing.

The ordering of scenarios generated through automation is random, and such an order does not ensure effectiveness in terms of objectives like fault detection and coverage. Besides customer inputs on priority, techniques to prioritize requirements and scenarios based on the primitives of the use case and activity diagrams are introduced in this work. A weighted sum of customer inputs as well as priority based on primitives is used to prioritize requirements and scenarios. Prioritization ensures early feedback to developers for bug fixing and further development.

Prioritization orders scenarios according to some criterion. However, testing exhaustively is not possible. Hence, there is need for selecting a subset of scenarios that best represents the complete set of scenarios. In this work, distance measures are used as basis for selecting a subset of scenarios based on their similarity. Another technique used is to cluster scenarios and select a representative subset from the clusters.

Different criterion are used besides random for both prioritization and selec-
tion. They include priority of scenario and type of scenario (main, alternate and exception) used for selection. The ordering and selection of scenarios are based on two criterion, namely, rate of fault detection and percentage of coverage. Average Percentage of Faults Detected (APFD) metric is used to calculate the rate of fault detection. Coverage is measured in terms of activities, transitions, basic path and additional path coverage criterion.