Slicing Technique for Test Path Generation in Concurrent Programs

A Thesis submitted

in partial fulfillment of the requirements for the award of degree of

DOCTOR of PHILOSOPHY

By

Vinay Arora

(951003001)

Under the guidance of

Dr. Maninder Singh
Professor
Computer Science and Engineering Department
Thapar University, Patiala
Punjab, INDIA

Dr. Rajesh Bhatia
Professor
Computer Science and Engineering Department
PEC University of Technology, Chandigarh
INDIA

Computer Science and Engineering Department

Thapar University, Patiala – 147004, Punjab, INDIA

August, 2017

Dedicated

To

The Almighty

Certificate

This is to certify that the thesis entitled **Slicing Technique for Test Path Generation in Concurrent Programs**, submitted by **Vinay Arora (Roll No. 951003001)** to Thapar University, Patiala, embodies the work carried out under our supervision; and that is worthy of consideration for the award of the degree of Doctor of Philosophy.

Dr. Maninder Singh
Professor
Computer Science and Engineering Department
Thapar University, Patiala
Punjab, INDIA

Dr. Rajesh Bhatia
Professor
Computer Science and Engineering Department
PEC University of Technology, Chandigarh
INDIA

Declaration

I certify that

a. The work contained in this thesis is original and has been done by myself under the general supervision of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in writing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

Vinay Arora

(Vinay Arora)

Acknowledgements

I would like to express a deep sense of gratitude to my supervisors, Dr. Rajesh Bhatia and Dr. Maninder Singh, for their constant guidance, help and continuous support throughout my research work. I sincerely acknowledge them for painstakingly reading my reports and helping me with their insightful comments at all stages of my research. Their discipline, hard work and attitude towards research always inspired me. Their enthusiasm has been a constant source of inspiration for me to attain my goal. I am highly thankful to them for the valuable time they spared for me.

I thank the doctoral committee members, Prof. Seema Bawa, Dr. Shalini Batra and Dr. M. D. Singh for sparing their valuable time to evaluate the progress of my research work. I also thank all the teaching and non-teaching members of the department for their kind help. I would also like to thank Prof. P. Gopalan (Director, Thapar University) and Prof. O.P Pandey (Dean Research) for providing me all the research facilities (tools and software) available in the organization. I am thankful to all of my friends for their presence and companionship which lit some dark and difficult moments in the course of my research. I am also thankful to Dr. Harish Garg for his valuable technical assistance in statistical analysis of the work. I express my gratitude to Dr. Alok Garg for providing hands on orientation of End Note, which has been quite useful for thesis writing. I thank Mr. Varinder Pal Singh for giving an orientation and kick start related to UML modeling tool Rational Software Architect (RSA).

Last but not the least, my acknowledgements would not be complete without a mention of my daughter Yevika, my wife Shruti and my parents, Sh. Kawal Jit and Mrs. Sudesh, who have always been a source of great strength, support and motivation for me. Without their constant encouragement, this venture would never have been possible.

Publications out of this work

Vinay Arora, Rajesh Bhatia, Maninder Singh, "A systematic review of approaches for testing concurrent programs," *Concurrency and Computation: Practice and Experience,* Wiley, Vol. 28, Issue. 5, pp. 1572-1611, 2016. DOI: 10.1002/cpe.3711
[SCIE Indexed, Impact Factor: 1.133, Citations: 02]

Vinay Arora, Rajesh Bhatia, Maninder Singh, "Synthesizing test scenarios in UML activity diagram using bio-inspired approach," *Computer Languages, Systems & Structures,* Elsevier, Vol. 50, pp. 1-19, 2017. DOI: 10.1016/j.cl.2017.05.002
[SCIE Indexed, Impact Factor: 1.615, Citations: 01]

List of Figures

List of Tables

List of Abbreviations

ASIS	Ada semantic interfaces specifications
ABC	Artificial bee colony
ACBP	All concurrent binomial path
ACO	Ant colony optimization
ACP	All concurrent paths
AD	Activity diagram
ANN	Artificial neural network
AO	Amoeboid organism
AS	Ant system
BFS	Breadth first search
BPEL	Business process language
CBT	Code-based testing
CCFG	Concurrent control flow graph
CCG	Concurrent composite graph
CFG	Control flow graph
CFN	Control-flow net
CQS	Concurrent queue search
CR	Combination reduction
CRD	Centre for reviews and dissemination
CSDG	Concurrent system dependence graph
DFS	Depth first search
DG	Dependence graph
DUN	Definition-use net
EA	Evolutionary algorithm
ECCFG	Extended concurrent control flow graph
ESYN	Extended synchronization
GA	Genetic algorithm
GUI	Graphical user interface
HAZOP	Hazard and operability
IOLES	Input output labeled event structure
ISTC	Interaction sequence testing criteria

ITM	Intermediate Testable Model
JMFD	Java multithreaded flow diagram
JPF	Java Path Finder
LSHB	Little strong happen before
LTL	Linear time logic
LTS	Labeled transition system
MBSE	Model-based System Engineering
MDA	Model-driven Architecture
MDD	Model-driven Development
MPI	Message passing interface
MSDG	M-S pair program dependence graph
MSPC	Message sequence paths criteria
NHST	Null hypothesis significance testing
OOCPDG	Object oriented concurrent program dependence graph
OSCI	Open SystemC Initiative
PDG	Program dependence graph
PDN	Process dependence net
PERT	Program evaluation and review technique
PIN	Process influence net
POC	Proof of concept
PSCL	Predicate sequencing constraint logic
RAG	Resource allocation graph
SD	Sequence diagram
SDN	System dependence net
SMMP	Shared memory multiprocessor
SR	Send receive
SRS	Software requirement specifications
SUT	System Under Consideration
S-V	Sequence/Variant
SYN	Synchronization
tCFG	Threaded control flow graph
TIRG	Thread interaction reachability graph
TLM	Transaction level modeling
tPDG	Threaded program dependence graph
UDG	Use case dependence graph

UML	Unified modeling language
VCT	Variable cache table
WAR	Write-after-read
WAW	Write-after-write
XCFG	Extended control flow graph

Abstract

A system is concurrent if it includes a number of execution flows that can progress simultaneously, and interact with each other. In this era of technology the concurrent systems are common in various application domains, *viz.* segregation of input-output processing from the back-end computation using multithreaded paradigm, client-server communication model having client-side and server-side computations, coordination among a large number of computing nodes in peer-to-peer services, *etc*. Synchronized implementation of the multiple execution flows in the concurrent software systems leads to new problems and introduces new design and verification challenges. This research work is a modest attempt to investigate in the domain of testing concurrent programs by classifying it into eight categories, *viz.* (i) reachability testing, (ii) structural testing, (iii) model-based testing, (iv) mutation-based testing, (v) slicing-based testing, (vi) formal method-based testing, (vii) random testing, and (viii) search-based testing. The following data/findings have been synthesized from the various research articles to form the basis of this work: (1) the approaches for testing concurrent programs; (2) the parameters for classifying various approaches into different categories; (3) the algorithms/techniques proposed or available under a particular approach for testing concurrent program; (4) the graphical representations used under a particular technique; (5) the test tools, prototypes and APIs proposed or available in this area; and (6) the subject systems or user programs used for testing concurrent programs.

The present study further investigates the problems of generating test scenarios from UML activity diagram using two bio-inspired algorithms, *viz.* amoeboid algorithm, and orientation-based ant colony algorithm. The first approach, based on the application of amoeboid organism algorithm, motivates to find out scenarios for concurrent section in an activity diagram. A similarity lies between the tubular veins like structure of *Physarum Polycephalum* and the test scenarios that might be generated by incremental traversal over the edges present in the similar tree type structure representing the permutations under fork-join node. Flux movements in *Physarum Polycephalum* generate articulated paths from the source node (root node) to sink node (leaf node).

The second approach, based on the application of ant colony algorithm, helps to generate the scenarios from a concurrent section of an activity diagram. A similarity exists between the pheromone-based path traversal by the ants from their nest to food source. The test paths that might be produced by traversing the edges present in the tree structure depicting the permutations under concurrency construct. In simple Ant Colony Optimization (ACO) approach, next hop node is decided by pheromone intensity along the edge between the current node and tentative next node. The issue arises when initial pheromone intensity along every edge between the current node and the nodes belonging to the set of probable next nodes is the same. Therefore, the ant cannot select the initial edge of the feasible path with bigger probability. An orientation factor has been introduced while computing the probability of selecting the next hop. Orientation-based ACO considers pheromone intensity as well as the angular factor between the current node and the next node.

Simulations have been performed using eight subject systems taken from the LINDHOLMEN data-set, four models, two each for both the proposed approaches, taken from real life student projects and five synthetic models. Null hypothesis significance testing (NHST) has been used as an inferential statistical method for deciding whether a well-specified hypothesis, identified as the *null hypothesis*, is to be regarded as true for a population from which a given set of data has been obtained by random sampling. The results obtained have been validated through statistical analysis which demonstrates that the proposed approaches are better than the existing ACO and Genetic Algorithm (GA) by a number of feasible test scenarios generated.

Table of Contents

Chapter 1

Introduction

The research in the field of testing of the concurrent program is being undertaken for more than 40 years. There has been a tremendous increase in the popularity of concurrent programs especially with the advancement of multi-core processors. The developers write a parallel code to improve the utilization of these processors that enhance the overall performance of the system. Shared memory is a predominant paradigm used for writing parallel code. Here, multiple threads of control form communication by reading and writing shared data objects. The shared-memory multithreaded code often faces a challenge from the bugs such as data races, atomicity violations, and deadlocks. It is always difficult to detect these bugs because the multithreaded code can demonstrate non-deterministic behaviors based on the scheduling of threads [1]. Further, the bugs may only be triggered by a small specific set of schedules. Thus, a challenge always lies ahead to build a reliable multi-threaded software. Concurrent programs are being used to replace the sequential programs as they make use of the true capabilities of multi-core architecture. There has been an immense use of multi-core systems and multithreaded paradigms, which asks for greater attention to the testing of concurrent programs. The field covers various domains, which include concurrency problems, testing approaches, techniques, graphical representations, tools, and subject systems. The research in the domain of testing concurrent programs can be broadly classified into eight categories, *viz.* (a) reachability testing, (b) structural testing, (c) model-based testing, (d) mutation-based testing, (e) slicing-based testing, (f) formal methods, (g) random testing, and (h) search-based testing [2].

Concurrency is defined as interleaving operation of processes on the CPU, which creates the illusion that these processes are operating at the same time. Concurrency intern implies the mutual interaction of the processes in parallel.

(Definition) Interacting Processes: Two processes P_i and P_j are interacting processes, if:

$$read\_set_i \cap write\_set_j \neq null \; or \; read\_set_j \cap write\_set_i \neq null$$

Where,

$read\_set_i = set\ of\ data\ items\ read\ by\ process\ P_i$

$write\_set_i = set\ of\ data\ items\ modified\ by\ process\ P_i$

The Table 1.1 describes the various methods through which process interaction can take place.

Table 1.1: Types of process interaction.

S.No.	Interaction	Description
1.	Data sharing	Shared data may lose its consistency if some processes update the data at the same time. Thus, processes must interact to decide when it is safe for a process to access shared data.
2.	Message passing	Exchange of information is processed by sending messages to one another.
3.	Synchronization	The processes must coordinate their activities and perform their actions in the desired order to achieve a common goal.
4.	Signals	The use of signals is made to communicate about the occurrence of an exceptional situation of a process.

1.1 Execution of a Concurrent Program

Let's consider an example to explain the concurrent program processing. In an airline reservation application, a number of agent terminals remain connected to a central computer system. The data is stored in a computer in a centralized fashion. All the agent terminals can access this data in real time. Execution of the application involves multiple processes. Each process services one agent terminal. Under such an arrangement, an agent at one terminal keys in the requirements of a customer, while other agents access and update the data on behalf of other customers. This perspective can be described in two ways: (a) High-Level Program, (b) Equivalent Machine Instructions.

Table 1.2: High level program and its equivalent machine instruction [3].

	High level program		**Equivalent machine instruction**
S_1	**if** *nextseatno* $\leq$ *capacity*	$S_{1.1}$	Load *nextseatno* in reg_k
		$S_{1.2}$	If $reg_k >$ *capacity* goto $S_{4.1}$
	then		
S_2	*allotedno* : = *nextseatno*;	$S_{2.1}$	Move *nextseatno* to *allotedno*
S_3	*nextseatno*: = *nextseatno* + 1;	$S_{3.1}$	load *nextseatno* in reg_j
		$S_{3.2}$	add 1 to reg_j
		$S_{3.3}$	store reg_j in *nextseatno*
		$S_{3.4}$	go to $S_{5.1}$
	else		
S_4	Display “sorry no seats available”	$S_{4.1}$	Display “sorry…..”
S_5	….	$S_{5.1}$	……

As is evident from Table 1.2, processes P_i and P_j share the variables *nextseatno* and *capacity*. Each process examines the value of *nextseatno* and updates it by 1, if a seat is available. Thus, a_i and a_j are identical operations. Statement S_3 in high-level language code corresponds to 3 instructions $S_{3.1}$, $S_{3.2}$ and $S_{3.3}$ that form a load-add-store sequence of instructions.

Table 1.3 describes three different executions of processes P_i and P_j when *nextseatno=300* and *capacity=300*. In **Case 1**, process P_i executes **if statements** that compare values of *nextseatno* with *capacity* and proceed to execute statements S_2 and S_3, which allocate a seat to it and increment *nextseatno*. When process P_j executes if statement, it finds that no seats are available. Thus, it does not perform any seat allocation.

In **Case 2**, process P_i executes **if statement** and finds that a seat can be allocated. However, P_i gets preempted before it can perform the allocation. Process P_j now executes if statement and finds that a seat is available. It allocates a seat and exits. The

variable *nextseatno* is now 301. However, when process P_i is resumed, it proceeds to execute instruction $S_{2.1}$ because it had ascertained the availability of a seat before it was preempted. Thus, it allocates seat numbered 301 even though only 300 seats exist! With *nextseatno*=301, the system should not lead to allocation of a seat. Hence, the final result is wrong here.

In **Case 3**, process P_i gets preempted after it loads 300 in reg_j . Now, both P_i and P_j allocate a seat each, however, *nextseatno* is incremented by only 1! Thus, cases 2 and 3 involve race conditions.

Table1.3: Race condition in airline reservation system [3].

Time instant	***Action of***		***Actions of***		***Action of***	
	P_i	P_j	P_i	P_j	P_i	P_j
1.	$S_{1.1}$	-	$S_{1.1}$	-	$S_{1.1}$	-
2.	$S_{1.2}$	-	$S_{1.2}$	-	$S_{1.2}$	-
3.	$S_{2.1}$	-	-	$S_{1.1}$	$S_{2.1}$	-
4.	$S_{3.1}$	-	-	$S_{1.2}$	$S_{3.1}$	-
5.	$S_{3.2}$	-	-	$S_{2.1}$	-	$S_{1.1}$
6.	$S_{3.3}$	-	-	$S_{3.1}$	-	$S_{1.2}$
7.	$S_{3.4}$	-	-	$S_{3.2}$	-	$S_{2.1}$
8.	-	$S_{1.1}$	-	$S_{3.3}$	-	$S_{3.1}$
9.	-	$S_{1.2}$	-	$S_{3.4}$	-	$S_{3.2}$
10.	-	$S_{4.1}$	$S_{2.1}$	-	-	$S_{3.3}$
11.	-	-	$S_{3.1}$	-	-	$S_{3.4}$
12.	-	-	$S_{3.2}$	-	$S_{3.2}$	-
13.	-	-	$S_{3.3}$	-	$S_{3.3}$	-
14.	-	-	$S_{3.4}$	-	$S_{3.4}$	-

The interacting processes need to coordinate their activities with one another to achieve a common goal. Sequential software systems are constituted of a single thread of execution. The testing process of a sequential system provides different input values to the system and investigates the behavior of the system under the given inputs. However, testing concurrent software is more challenging than testing sequential software since the behavior of a concurrent program not only depends on input values but also is affected

by the way the executions of threads are interleaved, *i.e.*, even under fixed input values, different interleaving of executions of threads may lead to different behaviors. Test generation for concurrent systems explores not only the input space to find a set of input values, which may trigger a bug (input generation), but also explores the interleaving space, to find the possible bugs (schedule generation). However, the exploration space being huge for real programs, it is not feasible to fully explore both input and interleaving spaces.

A concurrent program complies with *non-determinism,* which imposes a partial ordering for the occurrence of activities involved; and results into an uncertainty to the final sequence of occurrence of the events corresponding to the activities involved in the system under consideration. On the contrary, a sequential program inflicts a total ordering on the events or the activities involved in the system. Providing the same input data results into diversified paths, even on repetition of the concurrent program execution. Modus operandi for developing a sequential or concurrent program is almost the same, which include three main steps *viz.* requirement analysis and gathering, software design, and development. Concurrent systems can be designed by adopting the following main approaches:

- *Result parallelism*: It corresponds to the parallel behavior which is mainly based on the data structure used by the program.

- *Agenda parallelism:* It imposes a parallel behavior *w.r.t.* the design, which is based on the sequence of steps present in the requirement specification document for a specific computation.

1.2 Model-based System Engineering

Model-based System Engineering (MBSE) is considered as a formal application of the modeling process, which helps in the formulation of requirements, design, analysis, verification and validation activities that begin at the level of the conceptual design phase and continue throughout the software development life cycle. A model is said to be an approximation, representation, or idealization of the aspects related to structure, behavior, or operation that map with the corresponding real-world process, concept, or

system (IEEE 610.12-1990). Figure 1.1 depicts the sub-parts in Model-based system engineering.

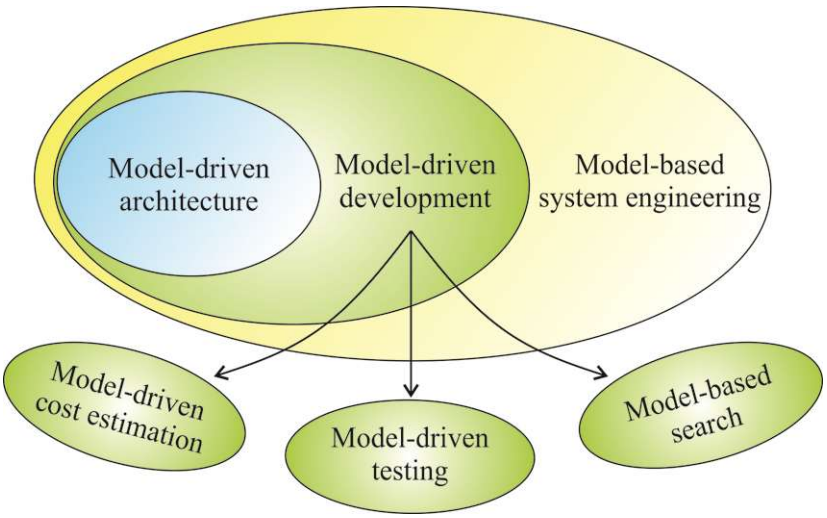


Figure 1.1: Sub-components in Model-based system engineering.

Model-driven Development (MDD) is a development paradigm that uses models as the prime object for the process of software development. The core idea for model-driven development is to depict software systems as a fusion of models that provide syntactic demonstration of domain knowledge, meta-models that express classes of models, and various transformations that render models to models or models to code of some specific language.

Model-driven Architecture (MDA) is a sub-set of the activities that come under MDD and is proposed by the Object Management Group (OMG). In MDA, the modeling and transformation languages are standardized by OMG [4]. MDA specifies a design approach for developing the software systems; and it incorporates a set of guiding principles for structuring the specifications expressed as models.

MDA implies the applicability of model-based approaches throughout SDLC, as it indulges the conceptual grounds related to requirement specifications and design models. Usage of models makes it feasible for model-based testing (MBT) to generate the test data, which is free from any specific implementation details at the initial design phase. Hence, MBT reduces the investment of resources in the context of time and effort. This motivates the researchers to use analysis and design models like Unified Modeling Language (UML) for test case/scenario generation. In software industry as well as in the

research field, UML is a well-established standard and is used for designing methodologies. UML constitutes a variety of depictions to cater with various aspects of system models [5].

Software testing can be undertaken at any time in the software development process depending on the development strategy being followed. In code-based testing, the test efforts are applied mainly after defining the requirements and completion of the implementation phase. In model-based testing, the testing process is applied in parallel with the development phase. The model-driven development provides the liberty to carryout testing processes at higher abstraction levels. It also exhibits code to model compliance through Model-Based Testing (MBT). The test cases are generated by MBT through the use of models communicating the system's expectations. Code-based testing helps to test what the code does, while MBT helps in testing what the code is supposed to do. Thus, it is complementary to the code-based testing. Apart from it, MBT is useful for some testing tasks when the code is not available [6]. With relevant test suites and test cases, better complexity management and comprehension of the system can be achieved with the model-based testing. In MBT, less time and effort are required because of an early start in SDLC. MBT's human-friendly abstractions for the significant aspects of the solution enhance the development productivity and quality.

While comparing MBT with code-based testing, the developer or tester collects the test scenarios from an actual program code. The **Code-based testing (CBT)** approach has certain drawbacks. Firstly, mere CBT cannot certify that the perceived working of software or a program is analogous to the intended behavior. Secondly, CBT can only start when real executable program or Proof of concept (POC) is available; and it is achievable late in the system development process. Thirdly, whenever there are certain changes in the executables, the whole test suite is required to be updated with new test cases/ or scenarios. Complementary to this, the specification-based testing is based on the required properties instead of any specific implementation. The specification-based tests can be developed as soon as Software Requirement Specifications (SRS) document is available (*i.e.*, before the first line of code) [7]. Figure 1.2 illustrates the involvement of models at various phases of SDLC.

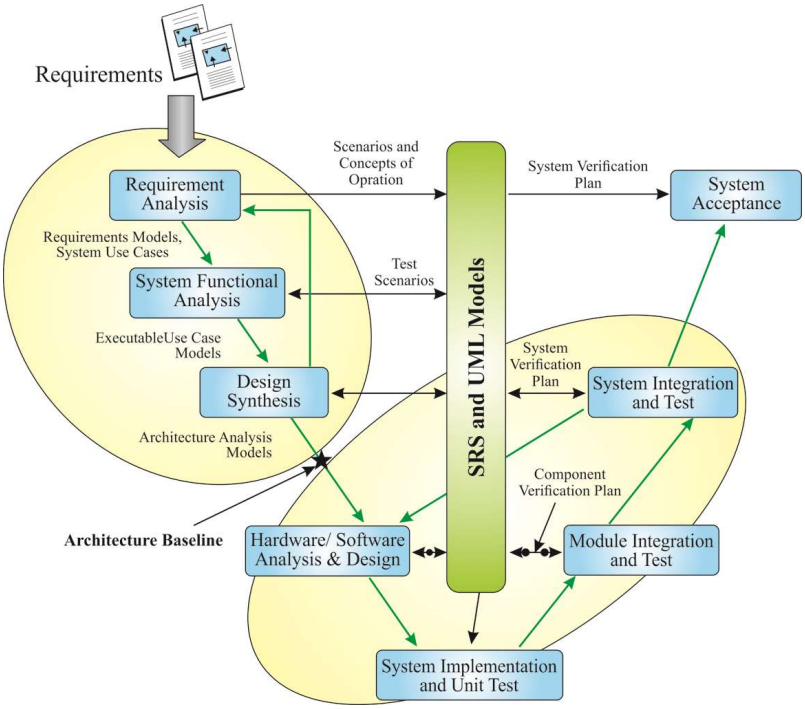


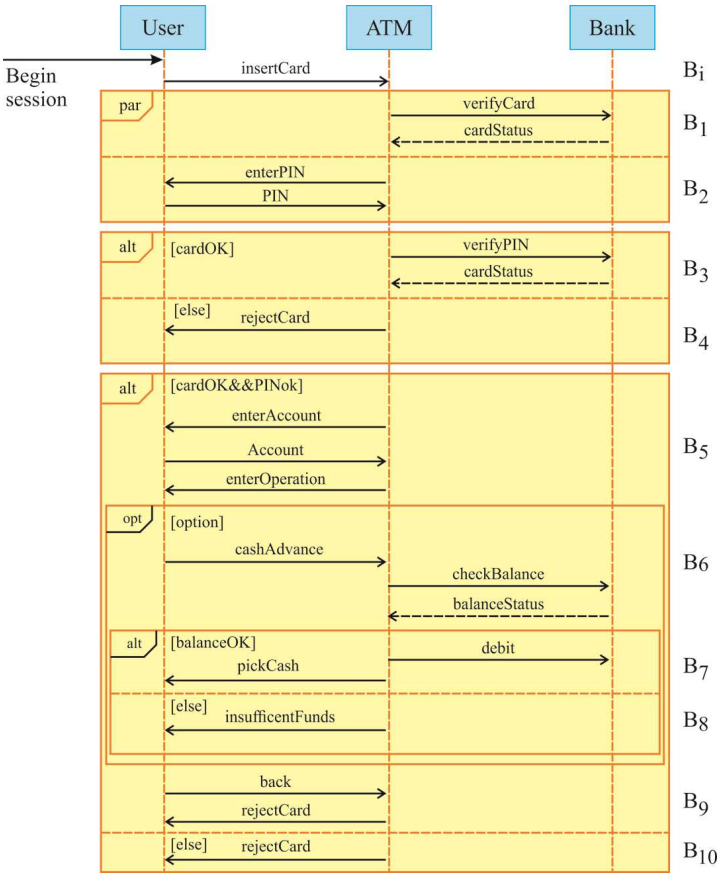
Figure 1.2: Involvement of UML models at various phases in SDLC.

The specification formalism, representing the system properties, behaves as a deciding factor for computing the effectiveness of an automated specification-based test case derivation method. In MBT, the system properties are explained as formal or semi-formal modeling language constructs like state-based formalisms *viz.* Finite state machines, State charts, UML state machines or State flow models. These formalisms have well-defined semantics, and these are easy to deploy. In a context when system model focuses on the functional requirements, the diagrams in MBT are used to get an executable by step-wise refinement [7].

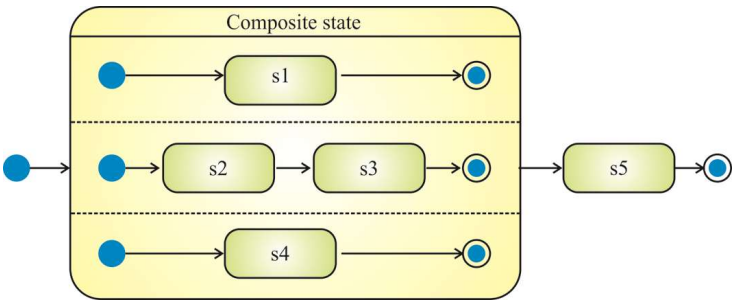
1.3 UML Behavioral Diagrams and Concurrency

This is an era of object-oriented software engineering. The researchers and developers are using UML as a standard for modeling the logical aspects of the program/system under consideration and developing its blueprints. A UML model provides standard and utility for the design approaches that exist for software/system development. UML lends the specific help in the specification, visualization, construction, and documenting models of software or the system. UML clearly depicts the structural and design aspects in such a manner that complies all the requirements specified in the SRS. Various views for a particular system can also be elaborated by the same [8].

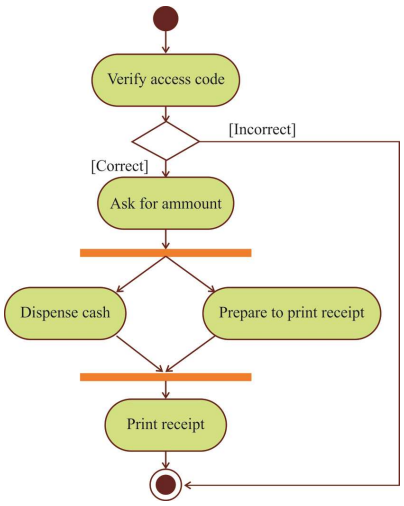
The behavioral diagrams, *viz.* sequence diagram, state machine diagram, and activity diagram [9] in UML are used for depicting the functionality of the System Under Consideration (SUT). The concurrent construct can be exhibited in the said diagrams for depicting the functionality, which is required to be executed in parallel or concurrently. Figure 1.3 highlights the concurrency element in the said diagrams.



(a)



(b)



(c)

Figure 1.3: Concurrency construct in (a) sequence diagram [10], (b) state machine diagram, and (c) activity diagram [11], [12].

There prevails a complexity in concurrent systems due to the non-deterministic interactions among the various objects or activities present in SUT. A Sequence Diagram (SD) is considered synonymous with an interaction diagram and depicts the scenario as a feasible sequence of messages that may exchange between the objects present in SUT. Due to static nature of lifelines in an SD, describing the scenarios will lead to complexity in case the modeling scenarios involve iterative or recursive constructs on a collection of participant objects. When a lifeline appears in an SD loop fragment, it refers to the same participating object across the iterations. In each iteration, if there is a need to access a different participant present within the collection, then participating entities must be depicted as separate lifelines [13], [10]. An SD represents the intra-instance interactions. Figure 1.3(a) represents an SD that represents a conditional entity by using alt and/or loop construct. A parallel or concurrent construct showing parallel behavior is depicted by using a par combined fragment.

To describe the interactions and the response messages sent to and from the entities, the state machine models, and activity diagrams can also be used as an alternative to an SD. In UML 2.0, state machines are the executable variants that resemble deeply in the functionality with its older versions [13]. Figure 1.3(b) displays the internal constructs of a state machine which are states, transitions, events, and activities. Transition means the flow of changing from one state or condition to another; events means the things that

initiate a transition, and activity relates to the response to a transition. Dynamic aspect in state machine diagram depicts the states that an object can attain during its whole life, and the events, which impact the state change with specific responses. In any particular state machine depiction, a node and edge denote state and transition respectively [8]. Composite states with two or more regions correspond to the concurrency construct in a UML state machine. Complexity with UML state machine increases with the increase in the system size, as it enhances the count of states and associated transitions exponentially [13]. There exist instances where the concurrent part in the state machine diagram makes it tedious to visualize the existing transitions among state combinations [14]. In an information flow, behavior or functionality indulge in the system can also be defined using Activity diagrams (AD); and it is considered as an appropriate visualization construct as compared to its counterparts discussed here. An activity diagram is a simple graph, in which a node corresponds to an action, data storage unit, or control flow element. To determine the execution of the actions involved, the flow of tokens across edges is considered. ADs are quite capable of describing the method bodies procedurally [13]. An AD records a wide-ranging structure/view for the sequential as well as concurrent constructs present in the SUT.

Activity diagram provides the sequence of actions that a particular process follows in a system under consideration. An activity diagram gives a work-flow sequence from initial point to the end point with all the details related to the decision paths that lie in the succession of events contained in the activity. The basic idea of an activity diagram is to model all the activities and their possible execution orders. An activity diagram is capable of demonstrating a sequential, iterative, and concurrent flow present in the execution of the activities involved in the process [15]. A concurrent flow of activities initiate multiple flows of activities simultaneously and at the same time. Interleaving among activities resulted into a permutation of activities, which result in a concurrent flow. Deterministic flow in an activity diagram constitutes sequential, decisional, and an iterative flow, whereas non-determinism appears due to the concurrent flow. Completion of all incoming control flows results into a solo control flow in a concurrent scenario.

1.4 UML Activity Diagram and Test Scenario

UML Activity diagram is mainly used to describe the business workflow of a system and functional aspect of the system under consideration. In the context of business work flow, the full functional scenarios can be derived from an Activity diagram, where each is a path having series of nodes from the *Start* to *End* activity.

(Definition) Activity diagram can be defined as an 8-tuple $AD = (AS, BR, MR, FK, JN, K, TS, as_0)$; where $AS = \{as_1, as_2, \dots as_n\}$ is a fixed set of action states; $BR = \{br_1, br_2, \dots br_m\}$ a fixed set of branches between action states; $MR = \{mr_1, mr_2, \dots mr_v\}$ a fixed set of merge operations; $FK = \{fk_1, fk_2, \dots fk_y\}$ a fixed set of forks; $JN = \{jn_1, jn_2, \dots jn_x\}$ a finite set of joins; $K = \{k_1, k_2, \dots k_w\}$ is a set having final states and end of the activity flows; $TS = \{ts_1, ts_2, \dots ts_z\}$ a finite set of transitions that satisfies $\forall ts \in TS, ts =< gc > ed \vee ts = ed$ where $gc \in GC$, $ed \in ED, GC = \{gc_1, gc_2, \dots gc_l\}$ is a set of guard conditions; $ED = \{ed_1, ed_2 \dots ed_s\}$ is a set of edges of the activity diagram; where the unique initial state is denoted by as_0.

The basic elements of a UML activity diagram are activity and transition. Where an activity node is considered as a state of doing some task and can be further segmented into various categories *viz.* start activity, end activity, branch activity, merge activity, fork activity, and join activity. Connectivity between activities is denoted as a directed line and termed as a transition. This can be classified as control flow, data flow, and object flow. An activity diagram depicts the complex sequences of activities, describing conditional as well as parallel behavior. Conditional behavior is outlined by a branch and a merge construct, while a parallel behavior is explained using a fork and a join node. A branch construct is having one incoming transition with several guarded outgoing transitions. In the case of parallel or concurrent processes, numerous arbitrarily ordered executions take place due to the interleaving among activity nodes under fork-join [16].

Formally, a test scenario is termed as follows:

(Definition) Test Scenario: Let an 8-tuple $AD = (AS, BR, MR, FK, JN, K, TS, as_0)$ is considered as a UML activity diagram. Here, TS denotes a set of test scenarios for an AD. $\forall ts \in TS, ts$ is a sequence of action states and transitions, *i.e.* $ts = as_0 ts_0 as_1 ts_1 \dots as_n ts_n k \wedge as_i \in AS \wedge ts_i \in TS \wedge k \in K, i = 1,2, \dots, n.$

Test scenarios play a major role in software development life cycle and can be developed by considering the design level activity diagram or its respective source code. Scenario generation using UML behavioral diagram has the edge over the code-based approach. First, we require fewer quanta of data to be processed alternative to large code. Second, we can discover test scenarios at the early stage of system (or software) development life cycle. It enables software analysts, developers or testers to build a useful test plan and development pattern before the start of the development phase [17]. Figure 1.4 demonstrates the basic paths in a UML activity diagram that can be considered as scenario here. Figure 1.4 (a) shows an activity diagram having one fork-join construct; Figure 1.4 (b) is its equivalent flow graph; Figure 1.4 (c) represents a set of scenarios that exist from start to end node for the activity diagram.

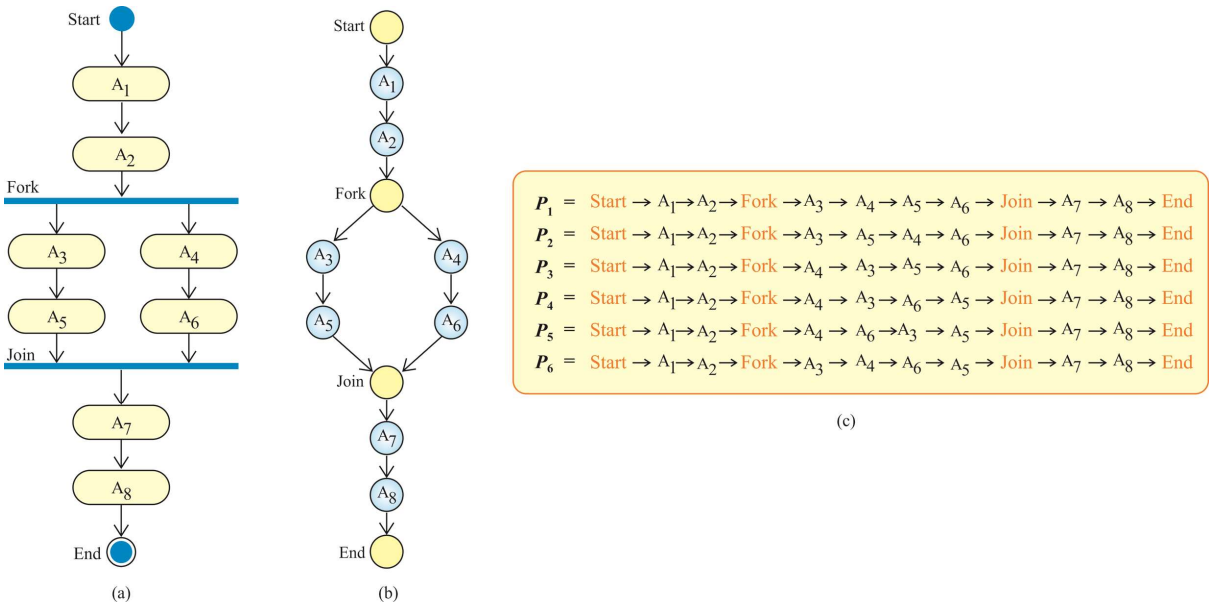


Figure 1.4: Paths in a UML activity diagram.

Testing software/program at cluster level becomes difficult when test cases are generated based on program source code. Development, as well as the testing process, becomes more efficient and effective when test scenarios or test cases are available early in the SDLC. Before generating the test cases and test drivers for a program or software under test, the test scenarios are generated using the available specifications, *etc*. The test scenarios generated manually consume ample units of time. Thus, either semi-automatic or fully automatic process for test scenario generation is desired. UML, a de-facto standard for modeling object-oriented software systems, is preferred for generating the

test scenarios. UML throws a minor challenge for generating the test cases as it provides a variety of diagrams for describing specific aspects of the software. Electing the right diagram for describing the appropriate aspect of the system poses such a challenge. UML diagrams are segregated by the structural or behavioral context of the system. An activity diagram describes the sequence of activities for the entities indulge in the control flow during the execution. A test adequacy criterion is efficiently achieved using activity diagram for generating the test scenarios. In scenario-oriented testing, scenario tests, or simple scenarios are used. Scenarios are helpful in connecting the functional requirements of SRS with the ones modeled using use cases. A scenario is considered as an instantiation of a use case, which takes a specific path through the model [18].

There exist many variations in the complexity of test scenarios when running for a series of use cases. Scenario testing is a well-established approach for studying the end-to-end delivery of data or control for complex transactions or events in the program under consideration. An activity diagram is used to depict the control flow of an operation with dynamic aspects of a group of objects presented in a single use case [18]. It also helps in representing the parallel activities with synchronization attribute involved in various activities [18].

1.4.1 Slicing for Test scenario generation

Slicing is termed as a process that involves computation of a set of statements corresponding to some values of interest, and that could get affected by some specific criteria. There are various techniques and algorithms proposed to test a concurrent program using slicing, *viz.* two-pass slicing algorithm, two-phase slicer, marking and unmarking based, vertex reachability, linear time logic (LTL), graph reachability, escape analysis, parallel algorithm, model reduction, operational semantics of concurrent flow chart language, data dependence analysis, dynamic slice computation, static slicing, context-sensitive slicing, slicing using trace witness, and slicing using meta-heuristic [2]. Slicing techniques are helpful in handling the complexity of programs which arises due to the large size of programs. Program slices are those statements of the program, which are relevant to a particular computation. Dynamic slices are generally of smaller size than static slices as these include only those statements of the program which are executed for a particular input data. Slicing techniques are quite useful in various aspects

of SDLC like software measurement, debugging, test-based development, and software maintenance [11].

Numerous techniques can be figured for analysis of a generic model of an SUT, which segments the analysis process into generation as well as testing of possible behaviors. A POC for the system under consideration is developed. Further, it leads to develop an illustration for getting the possible behaviors, which will be verified against the specification of acceptable behaviors, and report the noticeable deviations from the acceptable outcome or the exceptions. Young *et al.* presented a basic framework for analysis techniques and had taken the models as a program [19]. The model can be considered as the program augmented by auxiliary variables or control predicates in formal verification of the framework. The specification is marked by assertions in the program. Further, the representation of actual behavior is made up of a set of theorems derived from the program, axiom schemata, and rules of inference [19], [20], [21], [22], [23]. Considering UML model as a program, the fundamentals of slicing can be applied for extracting the desired set of elements/entities from the model under consideration.

1.4.2 Meta-heuristics for slicing

In the context of program/software development, the hard computing methodologies are quite cumbersome. Thus, a well-defined analytical model is required to overcome the problem. The soft computing techniques focus more on the interpretation of the system behavior than precision. They serve as a significant tool for many contemporary problems. Hence, they have been recognized as an effective alternative to the traditional approaches available for problem-solving. Artificial Neural Networks (ANN), Genetic Algorithms (GA), Fuzzy Logic Models (FLM), and Particle Swarm Optimization (PSO) [24] are some of the existing soft computing approaches. Among them, GA and PSO have been judged as quite potential and robust optimization tools. Meta-heuristics such as GA and ACO have their greater use in test scenario (or case) generation in conjunction with model-based testing [8]. Although slicing approach and meta-heuristics like ACO, GA, PSO, ANN, *etc.* are two different verticals, but still, both are providing benefits to each by narrowing the search space and enhancing the speed for extracting the statement of interest [25], [26], [27].

1.5 Objectives of the current research

The specific objectives of the proposed work are as hereunder:

a) To analyze and compare the existing techniques for finding the test path sequences in concurrent programs.
b) To propose an efficient technique for discovering the test paths for testing of concurrent programs.
c) To implement the proposed technique on concurrent programs.
d) To verify and validate the proposed technique.

1.6 Thesis Organization

The work is organized as follows:

Chapter 1: This chapter provides an overview of the domain of concurrency including basic concepts related to concurrency, a sample program, model-driven approach for software development as well as testing, concurrency constructs in UML behavioral diagrams, test scenario in UML activity diagram, slicing and meta-heuristics for generating the test scenarios.

Chapter 2: A comprehensive review of the available literature related to sub-domains under concurrency has been undertaken, which includes concurrency problems, testing approaches, testing techniques, graphical representations, tools, and subject systems. This chapter also presents the problem statement for the work considered in this thesis. This chapter outlines the basic concept of meta-heuristic, sources of inspiration in the context of applying meta-heuristic, design space of meta-heuristic, application areas of meta-heuristics, classification of meta-heuristic techniques, and a generic framework for a meta-heuristic.

Chapter 3: This unit provides the relevant details of the proposed amoeboid organism-based approach and basic motivation for applying the same to generate test scenarios from a UML activity diagram. It also provides the experimental work for the proposed amoeboid organism (AO) based approach. The activity diagrams from the existing

LINDHOLMEN data-set are taken as benchmark models for applying the existing ant colony-based algorithm and genetic algorithm. A comparison of existing ACO, and GA-based approach has been performed with the proposed AO-based approach using student projects and the synthetic activity diagrams having multi-join and a variety of control construct. Statistical analysis using t-test has been conducted to validate the proposed approach.

Chapter 4: This chapter deals with the modified ant-colony algorithm, and the same has been proposed for generating the test scenarios from UML activity diagram. Proposed work has been compared with ACO and evolutionary approach, being followed and used for generating the test scenarios. LINDHOLMEN data-set, student projects, and synthetic cases as taken up in the previous chapter have been considered for performing experimentation and making a comparison with the already existing techniques.

Chapter 5: This chapter concludes the study by highlighting the contribution made by the proposed research work. It also carries the useful directions for future work.

1.7 Thesis Contribution

This work contributes significantly in the following ways:

- The current research work extracts the relevant data from the primary studies on testing concurrent programs, and then synthesizes jurisprudentially to answer the following: (1) the approaches for testing concurrent programs; (2) the parameters for classifying various approaches into different categories; (3) the algorithms/techniques proposed or available under particular approach for testing concurrent program; (4) the graphical representations used under a particular technique; (5) the test tools, prototypes and APIs proposed or available in this area; and (6) the subject systems or user program used for testing concurrent program.

- Inferentially, this is a modest attempt to push forward the margins with regard to the gap analysis through which various novel research avenues for future explorations may be made possible for testing concurrent programs.

- Simulations of already existing meta-heuristic techniques for finding the test scenario in UML activity diagram have been done. A novel approach that uses the concept of amoeboid has been applied to get the advantage of natural phenomenon present for the proliferation and nutrients flow in the *Physarum*. Experiments have been conducted to prove the edge of proposed Amoeboid Organism (AO) -based algorithm over its peer approaches like the genetic algorithm, ant-colony meta-heuristic for computing the test scenarios in UML activity diagram.

- It has been proved that the proposed AO-based approach is better than the exact algorithm like depth-first search (DFS) for a large search space.

- Ant-colony optimization algorithm has been modified by applying an orientation factor for movement of an ant from the current node to the next probable node to generate a test scenario. Experiments have been conducted to prove the superiority of proposed orientation-based ant-colony algorithm over its peer approaches like the simple ant-colony optimization and genetic algorithm meta-heuristic for computing the test scenarios in UML activity diagram.

- The statistical technique of t-test has been applied for performing the analysis to validate the proposed techniques *viz.* Amoeboid organism, and Orientation-based ant-colony algorithm on benchmark models taken from LINDHOLMEN data-set, student projects, and synthetic models.

Chapter 2

Literature Review

This chapter aims to review the existing literature on the subject under study. The research questions formulated for the current study are stated in Table 2.1. There have been many studies carried out in the field of testing concurrent programs but with different perspectives and methods. These research studies have been reported in various leading journals such as Concurrency and Computation: Practice and Experience, IEEE Transactions on Software Engineering, Information and Software Technology, Electronic Notes in Theoretical Computer Science, Computer Languages, Systems and Structures, *etc*. Here, a sincere effort has been made to identify the gap in the research area, and understand the various issues, aspects, and methodologies related to the subject.

Table 2.1: Research questions

RQs		**Motivational factors**
1.	What is the present status of research in the area of testing concurrent programs?	To understand the domain related to testing concurrent programs.
2.	What are the different approaches used for testing concurrent programs?	To identify various approaches, techniques, algorithms, graphical representations used for testing concurrent programs.
3.	What techniques and graphical representations have been used for testing concurrent programs?	
4.	Which tools are available for testing concurrent programs? Under which testing approach do they fall? What is the citation frequency of these tools?	To list the tools, toolset, prototype or the application programming interface (API) available for testing concurrent programs under particular testing approach.
5.	Which subject system has been used under a particular testing approach and with what frequency?	To find the subject system used and the number of times a subject system has been used under a particular testing

RQs		Motivational factors
		approach.
6.	What are the research gaps and future directions in the area of testing concurrent programs?	To list the various sub-areas under a particular testing approach that can be explored further.

2.1 Approaches for testing concurrent programs

In software engineering, software testing includes testing techniques *viz.*, structural testing, model-based testing, mutation testing, formal method, search-based, random testing, and program slice-based testing, *etc*. Review work has thoroughly analyzed that although testing concurrent program has been widely discussed research domain, no standard categorization is available for testing approaches related to a concurrent program. Based on the framed RQs and the literature survey, testing approaches for concurrent programs can be broadly classified into ***eight*** different schools: reachability testing, structural testing, model-based testing, mutation-based testing, slicing-based testing, formal methods, random testing, and search-based testing. Segregation of all testing techniques into eight categories is based on the notion that these aforementioned schools are independent research domains although these domains and testing concurrent program can acquire the benefit from each other. Figure 2.1 refers to the categorization of approaches available for testing concurrent programs.

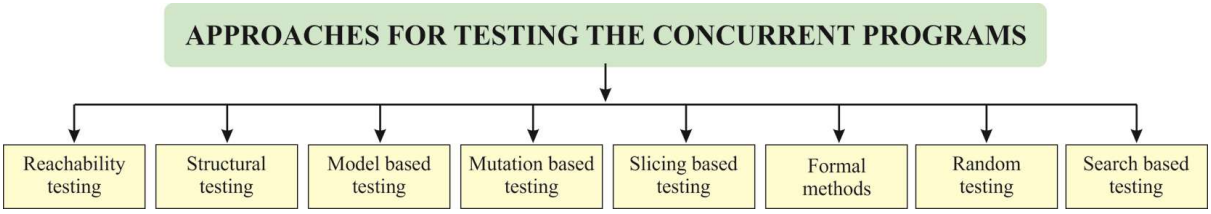


Figure 2.1: Approaches for testing concurrent programs.

The existing testing approaches in the domain of testing concurrent programs have been discussed in sub-section 2.1.1 to 2.1.8. Seminal papers have also been addressed where the researchers have contributed significantly for the first time.

2.1.1 Reachability testing

Reachability testing is the clubbed form of deterministic and non-deterministic testing techniques, which are used to test concurrent programs [28]. There are various techniques and algorithms proposed to test a concurrent program under reachability testing, *viz*. replay-based, analysis algorithm, language-based, race variant computation, apportioning technique, race table construction and reduction, happen-before, stateful reachability testing, synchronization (SYN) and extended synchronization (ESYN) sequence generation, *etc*.

In *replay-based approach*, Richard H. Carver *et al.* [29] described replay of test cases, where the program written in a particular language initially gets transformed into another program with synchronization events. Further, a deterministic testing was applied for an efficient regression testing of a concurrent program. An *Analysis algorithm* that addresses the tribulations like synchronization in processes, and error detection in the synchronization structure, was proposed by R.N. Taylor [30]. The author addressed a method for constructing a tool to ensure that the system will never enter in an infinite wait, and also ensured the absence of any undesirable parallelism. The only static analysis was used in this work [30] with no properties of an actual Ada program, whereas later works [31], [32] provided a run time analysis and also dealt with an actual Ada program. For solving the problems involved in the deterministic execution of a concurrent Ada program, Richard Carver and K. C. Tai [31], [32] described the *language-based approach*. The system presented by these researchers was free from any probability issues and can be used as a high-level design for an implementation-based Ada testing tool. The presented approach was found to be comparatively difficult and costly to implement, as it needs access to the underlying operating system.

An algorithm to compute *race variants* using read/write (RW) operation in concurrent programs was provided by Gwan-Hwan Hwang *et al.* [28]. However, the effect of non-synchronization events was not taken into account. Kuo-Chung Tai [33] derived race variants for send-receive (SR) sequences and proposed a non-deterministic and prefix-based algorithm for reachability testing of asynchronous message passing program. Further, the enhancement was given by Yu Lei *et al*. [34], which reduced the complexity as well as redundancy caused by totally ordered SR sequences. Reduction in the

complexity was due to the single-time execution of every partially ordered path. A scheme for testing java monitors through *deterministic execution* was explained by Craig Harvey *et al.* [35]. The method constitutes four steps viz. identifying pre-conditions, constructing a sequence of calls, implementing a sequence, and execution & comparison. Craig's method was based on previously proposed Brinch Hansen's method [36] for Pascal monitors and provided an extension at the first step named identification of suitable pre-conditions. Based on the classification of program points as local and global, Sridhar Iyer and S. Ramesh [37] proposed an *apportioning technique* for safe and efficient reachability analysis of concurrent programs. Proposed technique modeled the set of all the possible execution sequences by state transition graph known as a reachability graph.

For reachability testing using race table, Yu Lei and Richard H. Carver [38] presented an approach to find the sequence of events for the concurrent programs. Researchers have used semaphores to synchronize the operations on shared data. In the given approach, the researchers presented two types of semaphores, *viz.* counting and binary. Space-time diagram described the call and completion sequence for a concurrent program. The algorithm defined by the researchers constructs a race table to derive the race variants that can be used to derive call and complete sequence using prefix-based testing. Further, improvement was reported by Yu Lei *et al.* [39] that described a new strategy for reachability testing, in which algorithm guaranteed the single traversal of each partially ordered sequence, without saving any SYN-sequences. Another method proposed by Yu Lei *et al.* [40] for reachability testing that uses Sequence/Variant graph and an algorithm to compute race variants can be applied to several commonly used synchronization constructs. This method also constructs a race table for various synchronization sequences. However, it was unable to detect the missing sequences (sequences that are valid according to specification but are not allowed by the implementation).

The development of load balancing techniques for distributed reachability testing was enhanced by Richard H Carver *et al.* [41] that provided the design and implementation of a distributed *reachability testing algorithm* for a cluster of workstations. Different test sequences were executed concurrently by diverse workstations without having any synchronization and duplication of the sequence among workstations. The algorithm uses round-based and randomized work stealing protocol for dynamic load balancing. Simone

R. S. Souza *et al.* [42] proposed a technique, which validates concurrent programs using a combination of coverage criteria and reachability testing. Coverage criterion was used to select test cases and to determine the execution of new synchronizations, whereas, reachability testing was used to select appropriate synchronizations. This combined approach enhances the test coverage and also reduces the count of generated sequences. A *combination reduction* (CR) technique to select SYN-sequence in reachability testing was presented by LU Chao *et al.* [43]. This technique was based on the combination testing theory, where the count of SYN-sequences was reduced without decreasing the test effectiveness. For reducing the count of SYN-sequences, several steps *viz.* SYN-sequence generation for concurrent programs, race table construction, and race table reduction were followed, where the size of race table determined the count of test cases.

The work reported by PU Fangli *et al.* [44], defined a relation named *little strong happen before* (LSHB) for concurrent programs to select the reduced number of SYN sequences with high practicability. An approach for automatic selection of input that guides the reachability testing for statement coverage was proposed by Gwan-Hwan Hwang *et al.* [45]. In this approach, a sequence of path conditions was derived from the *SYN sequence*; and these collected path conditions were used for developing a constraint solver for deriving new inputs, and to find dead codes in concurrent programs.

For reachability testing, Shuang Quan *et al.* [46] illustrated a graph-based approach, where a set of *synchronization sequences* were obtained from the graph that represented beginning/end and synchronization events present in the program having single synchronization object. Execution models for commonly used synchronization constructs under timestamp assignment were provided by Richard H. Carver *et al.* [47] that use asynchronous message passing, synchronous message passing, and semaphores and locks.

Based on *ESYN sequence* that includes synchronization as well as non-synchronization events, Xufang Gong *et al.* [48] explained a technique under the reachability testing to test multi-threaded java program in which Java multithreaded flow diagram (JMFD) was used to compute ESYN sequences. This approach can be optimized further as the count of generated ESYN sequences is too large to cover. For reachability testing of a single monitor and its test threads, Richard H. Carver *et al.* [49] provided an enhanced approach

named *stateful reachability testing with variant and state pruning*, for storing and recognizing the visited states. As reachability testing is stateless testing technique, multiple visits of the program's state space have created the problem of sequence explosion. Reachability testing for the internet-based concurrent program that reduces race variants and increases the feasibility was presented by Fangli Pu *et al.* [50]. Feasibility strategy for reachability testing includes the steps, *viz.* finding happen-before relationship between race receive events, finding race variants and constructing race table, prefix based testing with race variants, and exercising complete SYN sequence. A vector timestamp was used to determine the happen-before relations between race receive events of synchronization pair. A class library named *Modern Multithreading* was described in the literature [51], [52] for Java and C++ that constitutes thread and synchronization classes to support testing and debugging of a concurrent program with shared variables, semaphore, asynchronous message passing, synchronous message passing, and monitors. For non-deterministic concurrent programs with an infinite number of possible interleavings, Che-Sheng Lin *et al.* [53] proposed an approach that performed state-cover testing and included the generation of partial order graph and event compression. This technique has bypassed the need for any static analysis; and therefore, resolved the problem of exhaustive testing.

2.1.2 Structural testing

Structural testing of the program is an approach that includes coverage of nodes, edges, and paths of a graph representing flow in a specific program. There are various techniques and algorithms proposed by research community for testing of a concurrent program using structural testing approach, *viz.* coverage-based, path analysis, path finding, combination algorithm, constraint-based, concurrent state graph, post-mortem method, LTS reduction, hot spot prioritization and topological sort, dataflow sets, all concurrent binomial path (ACBP), algorithm accept and recursive function propagate.

In *coverage-based* approach, Richard N. Taylor *et al.* [54] outlined the structural coverage criteria and coverage criteria hierarchy, where the researchers have presented a concurrency graph for finding out the path and the coverage criteria. Coverage criteria include concurrency state coverage, state transition coverage, and synchronization coverage. One of the major limitations foreseen in their approach was the application of

a concurrency graph for static analysis of small programs only. Parallelizing compilers converted sequential programs to parallel form and based on the parallelizing compiler, the seminal paper of Mary Jean Harrold *et al.* [55] presented the first technique with data flow tester that used program dependence graph (PDG) to determine test case coverage. In their work, authors presented a system for reengineering and retesting of programs in a shared memory multiprocessor environment, where probes were inserted into the parallel stream that resulted in an extremely compact program trace.

A *path analysis approach* for structural testing of concurrent programs was proposed by R-D Yang *et al.* [56], [57] where execution behavior of a concurrent program was obtained by a concurrent path model. In the proposed approach, a static structure of a concurrent program was modeled by flow graph and dynamic structure by rendezvous graph. A concurrent path analysis technique with *combination algorithm* and constraint collection algorithm was proposed by Jun Yan *et al.* [58] for generating the test paths from business process language (BPEL) program. BPEL program initially gets converted into an extended control flow graph (XCFG) that further generates the sequential test paths, which finally generate concurrent test paths. For the first time, the decisive paper of Maria A. S. Brito *et al.* [59] presented the concepts of experimental software engineering for the definition, conduction, and analysis of the testing criteria for concurrent programs. The researchers have explored the family of structural testing criteria for message passing concurrent program to find the cost, effectiveness, and strength of various testing criteria. Def-use testing is a form of path testing, which mainly focuses on the variables, *i.e.*, where in the program a variable receives the value, and where in the program the value received is used or referenced in the calculation. The first algorithm for *finding all-du-path* coverage in shared memory parallel programs was presented in [60], [61]. The work presented by C. S. D. Yang *et al.* [60] identified the issues in providing all-du-path coverage and usage of depth first search (DFS) or DT-IT approaches to find all-du-paths for shared memory parallel programs.

Using *Constraint-based* approach (which arises from an algebraic reformulation of intra-procedural methods); a framework for an inter-procedural analysis of fork/join parallel programs was presented by Helmut Seidl *et al.* [62]. Analysis of bit vector as well as non-bit vector problems was performed using the proposed framework. A technique named *annotated labeled transition systems* (ALTS) was reported by Pramod V. Koppol

et al. [63] for incremental reachability analysis. Strategies discussed for incremental structural testing included bottom up incremental testing, and incremental testing using program slices. Various synchronization methods were included under incremental coverage criteria, and one of the major advantages of their method was the early detection of faults in software development life cycle (SDLC) for concurrent programs. Parmod V. Koppol *et al.* [64] presented *LTS Reduction* for structural testing of a concurrent program where processes were integrated into a bottom up style. To reduce the size of a test suite, an enhancement was presented by Samira Tasharofi *et al.* [65], which used a partial order reduction technique for pruning the SYN sequences. Using *concurrent state graph*, G. Cantone *et al.* [66] represented the transformation of concurrent and messaging Ada tasks using D-graph. These transformations reduced the structural complexity of the generated graphs.

To determine the implicit communication, time stamp-based *Post-mortem method* has provided the coverage measure for evaluating the progress of testing activity in the concurrent programs [67]. To overcome the difficulty of inter-thread communication, a family of structural testing criteria was proposed for semaphore-based multithreaded programs. For structural examination, the information related to the control, data, communication, and synchronization was used. Based on *hot spot prioritization and topological sort*, W. Eric Wong *et al.* [68] proposed a method for test sequence generation, which resulted in a small set of test sequences. For parallel programs with lexical constructs, Dirk Grunwald *et al.* [69] described a *data flow framework* to compute the reaching definition information with explicitly specified lexical constructs. Work presented by the authors addressed the cyclic synchronization structures where an equation for computing the reaching definition information was executed to handle synchronization. Juichi Takahashi *et al.* [70] proposed *All concurrent binominal path* for high coverage rate and efficient exploration of concurrency related bugs. Concurrent module flow graph and ConTest was used for efficient coverage analysis.

Michael Factor *et al.* [71] listed a systematic evaluation of program-based coverage criteria for concurrent programs. These programs combine sequential and concurrent aspects where usage of more restrictive partial order enhanced the overall process of evaluation. Comparison of coverage criteria for a concurrent program using mozilla data race bug was presented by Shan Lu *et al.* [72]. For the definition of testing criteria,

S.R.S. Souza *et al.* [73] proposed a test model of control and data flows, which provided a measure to evaluate the progress of testing activity in concurrent programs. Further, the extension was provided by Paulo S. L. Souza *et al.* [74] with additional features like collective communication; non-blocking sends, distinct semantics for non-blocking receives, and persistent operations for revealing errors that were mainly related to inter-process communication. A comparison between the generic method for statistical testing (GMST), statistical usage testing (SUT) and exhaustive testing (ET) was provided by Miroslav Popovic *et al.* [75]. Where, ET method was used to discover hidden faults; SUT was applied to find deeply hidden faults on most frequently executed paths (with limited testing effort), and GMST was applied to find out the hidden faults on the most probable path and with uniform path coverage depth. Concurrent module flow graph and *all concurrent paths* (ACP) were used by Hideharu Kojima *et al.* [76] to execute white box testing for a concurrent program. The approach given by researchers has reduced the count of test cases of ACP and measured the coverage of ACP. To expose untested thread interleaving, a coverage-driven testing tool called Maple was presented by Jie Yu *et al.* [77]. This tool has enhanced the interleaving coverage by memorizing tested interleaving. As compared to random and systematic testing tools, this approach avoided testing of the same thread interleaving across different test inputs.

2.1.3 Model-based testing

Model-based testing is defined as the application of model-based design for designing and executing artifacts to perform software testing. There are various techniques and algorithms proposed to test concurrent programs through model-based testing approach, *viz.* Flow back analysis and Incremental tracing, Branch and bound techniques, Path generation algorithm, Co-path generation algorithm, Test and Debugging algorithm, Race and inconsistency detection, Algorithm for generating sub event graph, Change identification and test selection, Value-schedule-based testing, Concurrent queue search (CQS), Slicing based, Dependence analysis, UML state machine, ANDES, ExitBlock algorithm, Predicate coverage and use case sequence algorithm, Message sequence path with BFS and DFS, Labeled event structure and Formal specifications, Bottleneck detection approach, Object-Z, Labeled transition system (LTS), structured-object-based-formal language (SOFL) specifications, Transformation-based and Scenario-oriented test case generation, Trace visualization, Genetic algorithm, *etc*.

For parallel programs running on shared memory multiprocessor (SMMP), an integrated debugging system was provided by Barton P. Miller *et al.* [78]. To provide information on causal relationships among events, *flow back analysis and incremental tracing* was used, without re-executing the program for debugging. For searching the state space in concurrent systems, K. L. McMillan *et al.* [79] provided the *branch and bound* technique, which was based on unfolding the Petri nets. This unfolding approach has avoided the exponential explosion of the states, which has resulted from a total ordering of the transitions. In another approach [80], Co-paths on the event interaction graph (EIAG) that satisfied the interaction sequence testing criteria (ISTC) were generated as test cases, using *Co-path generation algorithm*. Interactions like synchronization, communication and wait were represented through event graph. This approach has reduced the insufficient and overlapping test cases. Co-paths generated on EIAG have been used for detecting the dead concurrent statements.

Test and debugging algorithm was reported by Suresh K. Damodaran-Kamal *et al.* [81], which focused on race detection in message passing parallel programs through space time diagram and task graph. Controlled execution has explored all the possible scenarios for the races. For *race and inconsistency detection,* an extended UML activity diagram was provided by Bin Lei *et al.* [82] that tagged additional data operation information with an activity diagram. Test cases were generated by traversing the path in an extended activity diagram. The program under test was instrumented by data operations tagged in an activity diagram. Further, an instrumented code was executed, and the execution traces were checked for the data race and inconsistency in the system. A tool named *toc4j* (testing of concurrency for java programs) analyzed the traces by offline execution and saved the computational cost associated with the runtime analysis. Xiaoan Bao *et al.* [83] generated test cases for concurrent programs from *sub event graph*, which were obtained as part of full event graph that represented a concurrent program. Using an extended concurrent control flow graph (ECCFG), a technique for *change identification and test case selection* was provided by Atifah Ali *et al.* [84], where sequence diagram and class diagram were used to construct the ECCFG. Baseline and modified ECCFGs were compared to find the altered nodes and arcs. To obtain the regression test set, altered nodes and arcs were supplied as input to test case selector and generator for the test case selection, and generation respectively.

Based on the static and dynamic analysis, Jun Chen *et al.* [85] presented the *value schedule-based* testing technique for a concurrent program, where static analysis was based on SOOT, and dynamic analysis was based on java path finder (JPF). The technique proposed by these researchers was integrated into value schedule-based testing technique, which provided an improvement over JPF. Generating exhaustive test cases for concurrent programs is exponential in size. Therefore, *CQS algorithm* [15] was considered as one of the efficient search algorithms for generating adequate test cases. CQS handled the random nature of concurrent tasks and uncovered the errors related to data safety and casual ordering among concurrent processes. Test sequences as generated by CQS were superior as compared to DFS and BFS search algorithms. Based on *temporal sequencing* of events and their data dependency, a symbolic test generation algorithm was proposed by PengWu *et al.* [86]. Here, the researchers have used the predicate sequencing constraint logic (PSCL) to define the test purpose, which acted as an input for the symbolic test generation algorithm. By temporal sequencing in test purpose, slicing was performed; and the test cases generated using data dependencies were defined for test purpose. Research reported by Matthew B. Dwyer *et al.* [87] claimed that by the year 2006, no systematic study, having the cost/benefits analysis of slicing technique for model reduction was available for the realistic systems. Arthorn Luangsodsai *et al.* [88] reported a technique for slicing state chart using And-Or dependence graph, which further includes interference dependencies. To find out the precise set of statements in a concurrent program, a *dependence analysis* algorithm was presented by Zhenqiang Chen *et al.* [89]. This algorithm was based on concurrent PDG and concurrent control flow graph (CFG) and was more precise as compared to the previous approaches that were proposed to calculate the dependence set of statements. Soon-Kyeong Kim *et al.* [90] presented an approach for generating test cases from *UML sequence diagram*. Initially, a class diagram followed by an enhanced state chart diagram was developed, which further got converted into symbolic analysis laboratory (SAL). Finally, SAL was used to generate the test sequences, which got executed by a tool named ConAn. A tool named ANDES that allowed performance evaluation at model level was addressed by Joao Paulo Kitajima *et al.* [91]. ANDES was based on synthetic programs and developed to support the evaluation of different mapping strategies for the parallel programs and architecture.

For a given input, to enumerate all the possible behaviors of a section in a multithreaded program, Derek L. Bruening *et al.* [92] described an algorithm called *Exitblock* that guarantees to test all program behaviors, while considering only the possible schedules of synchronized regions. In the proposed approach, threads were represented using a tree structure. Santosh Kumar Swain *et al.* [93] proposed a *predicate coverage and use case sequence algorithm*, where use case dependence graph (UDG) was used to generate use case dependency sequences. Respective sequence diagram was constructed by using the generated dependence sequence, which further generated test sequences by developing concurrent control flow graph (CCFG). The researchers also detected the faults related to interaction, message sequence, use case dependency, and synchronization. Monalisha Khandai *et al.* [10] gave an approach, where initially a sequence diagram was converted into an intermediate representation known as the concurrent composite graph (CCG). Further, this CCG was traversed using BFS and DFS, taking *message sequence paths* criteria (MSPC) to generate test cases. The test sequences were helpful for detecting interaction scenario, and the operational faults. In another approach [94], input output *labeled event structure* (IOLES) was used to specify the formal behavior of a concurrent program and to generate the test cases. Further, this was proved in research that the generated test cases were complete for co-ioco, where co-ioco is an extension of Input Output Conformance (IOCO) relation. The proposed system has provided aid for accommodating concurrency with I/O enabled system models.

For the model driven performance *bottlenecks detection*, Vahid Garousi [95] used a program evaluation and review technique (PERT) with unified modeling language (UML) sequence diagram, and interaction overview diagrams (IODs). Primarily, UML sequence diagram was used for constructing CCFG. Afterward, CCFG, IOD, and profiling information were used for constructing PERT diagrams. Finally, critical path analysis using PERT diagram was performed to identify the performance bottleneck. This technique has prevented the construction of specific performance model, like layered queuing network, which consumed huge effort for performance analysis. A formal model of java concurrency using the *object-Z specification* language was presented by Roger Duke *et al.* [96]. The researchers have listed the usage of the proposed model to explore the generation of test suites for the concurrent system implemented in java as future work.

Scott D. Fleming *et al.* [97] used *LTS* for internal and external representation of a multi-threaded program, and LTS based tool that has modeled the interactions for providing aid in the debugging process. Yuting Chen [98] presented a methodology where conditional data flow diagram (CDFD) of *SOFL specification* of a concurrent program was used to derive set of specification traces. Further, these specification traces were covered to generate test cases, and executed repeatedly to cover all paths in a concurrent program. Work reported by Chang-ai Sun [99], provided a *transformation-based and scenario-oriented test case generation* approach, where the UML activity diagram was transformed for generating Binary Extended AND_OR Tree (BETs). Further, test scenarios were generated using BETs, and an algorithm, which satisfied the concurrency coverage criteria.

Katharina Mehner *et al.* [100] extended UML by stereotypes for adding the Java specific modeling elements. Sequence diagram was used for *visualizing the traces* of the program, and for identifying the lifeness error. UML collaboration diagram was used to identify the reasons for the deadlock. Katharina Mehner [101] presented a UML-based environment for visualizing and debugging a concurrent Java program. This environment has incorporated the tracing-based deadlock detection and analysis technique. Messages or calls that triggered the changes were not shown in this program. Peter Mehlitz *et al.* [102] described the JPF; and Klaus Havelund *et al.* [103] translated Java programs into PROMELA models. These models were checked using SPIN model checker, for finding the deadlocks and violations from the stated assertions. Deadlocks, local assertion violations, and resource leaks in a parallel program written using message passing interface (MPI) were detected by model checking using In-situ partial (ISP) order [104]. Based on the analysis of design representations like UML and MARTE profile, a method for automated verification using the *genetic algorithm* was described by Marwa Shousha *et al.* [105]. The proposed method has reduced the need for complex tooling, and additional modeling, which was previously required for UML-based development.

2.1.4 Mutation testing

It is a method of software testing in which a program or source code is deliberately manipulated, followed by a suite of testing against the mutated code. There are various techniques and algorithms proposed to test a concurrent program using mutation testing,

viz. Mutant coverage algorithm, Concurrency failure detection, Fault-based testing, Binary set algorithm, Mutation analysis, Deterministic execution and mutation-based, Hazard and operability (HAZOP), Pseudo-code algorithms, Selective Mutation, *etc*.

The problem of verification for a concurrent program using Open SystemC Initiative (OSCI), SystemC mutant, Transaction Level Modeling (TLM) 2.0 library, and SystemC Runtime verification toolbox, was targeted by Alper Sen *et al.* [106]. Here, verification of the program was performed by using *mutation testing-based coverage matrices*. Using mutant monitors, various methods for *detecting concurrency failures* were proposed by Brad Long *et al.* [107] that verified the correctness of reproducible components of a concurrent program. The first work related to the generation of test purposes through specification mutation was proposed in the seminal paper by Aichernig BK *et al.* [108]. Researchers have presented a technique to generate the *fault-based test cases* for the concurrent systems, in which the faults were injected into the system under test to generate the test purposes. The *Binary set algorithm* was discussed by Shady Copty *et al.* [109] for pinpointing the location of a fault in a concurrent program. Binary set algorithm has provided an enhancement over the modified DD algorithm, which was used for searching the set having changes. To exercise the concurrency and synchronization available in a concurrent program, Marcio Delamaro *et al.* [110] proposed a set of mutant operators for the Java programming language. Mutation testing was successfully applied at the unit level as well as integration level testing of a concurrent program. Based on the notion of an SYN-sequence that characterizes the communication and synchronization in a concurrent program, Richard Carver [111] described an approach that combines *deterministic execution and mutation based* testing to test and debug a concurrent program.

Sunwoo Kim *et al.* [112] applied *HAZOP* to generate mutation operator in Java syntax definition for generating a complete set of mutation operators. For measuring the testability of a concurrent program without using the probes, an approach that inserts the concurrency related faults was proposed by Sudipto Ghosh [113]. For comparing the different testing strategies for testing the concurrent Java programs, a set of the concurrent mutation operators was provided by Jeremy S. Bradbury *et al.* [114]. With a motivation to improve the quality assurance techniques, the researchers [115] compared the concurrency testing with ConTest and model checking with JPF and concluded that

the ConTest was comparatively efficient as this can kill a mutant in less average time than JPF. Milos Gligoric *et al.* [116] presented a framework that reduces the time for the mutant execution in multithreaded code, whereas Leon Li Wu *et al.* [117] proposed another approach using a generic mutation testing framework. In the proposed approach, the researchers evaluated that the new set of first and second order synchronization centric mutation operators were more effective and applicable in mutant.

Rodolfo Adamshuk Silva *et al.* [118] introduced and categorized the mutation operators for message passing parallel program, like MPI. Madanlal Musuvathi *et al.* [119] used the CHESS tool for systematic testing of a concurrent program. Pseudo-code algorithm by Milos Gligoric *et al.* [120] provided a framework for reducing the time for mutation testing of multithreaded code using a tool named MuTMuT. A seminal paper by Milos Gligoric *et al.* [121] presented the first study on *selective mutation* for the concurrent mutation operators. This selective mutation has accelerated the process of mutation testing by applying only a subset of mutation operators.

2.1.5 Slicing-based testing

Program slicing is defined as the computation of a set of program statements that may affect the values of a variable at some point of interest, referred to as a slicing criterion. There are various techniques and algorithms proposed to test a concurrent program using program slicing, *viz.* Two pass slicing algorithm, Two phase slicer, Marking and unmarking-based, Vertex reachability, Linear time logic (LTL), Graph reachability, Escape analysis, Parallel algorithm, Model reduction, Operational semantics of concurrent flow chart language, Data dependence analysis, Dynamic slice computation, Static slicing, Context sensitive slicing, Slicing using trace witness, *etc*.

In 1993, slicing concurrent program was addressed for the first time by Jingde Cheng [122]; and the main focus was given to the graphical representation of a concurrent program. Here, process influence net (PIN), process dependence net (PDN), definition-use net (DUN), and control-flow net (CFN) were used for depicting various dependencies. Using dependencies in PDN and PIN, the problem for finding the slice was reduced to the vertex reachability problem and was proposed to be addressed using BFS and DFS. For finding the static and dynamic forward slices, researchers have used

the *vertex reachability* in stand alone fashion and with program execution history both. For slicing of a concurrent object-oriented program, Jianjun Zaho *et al.* [123] extended the program dependence representation by a system dependence net (SDN). Using SDN, slicing of a concurrent program was simplified to *vertex reachability* problem again, where initially an SDG got converted into an SDN; and a two pass algorithm was applied to compute the program slices. Jianjun Zaho [124], [125] represented a concurrent Java program by multithreaded dependence graph and applied a *two pass slicing algorithm* to compute the static slice. Proposed technique initially converted a Java program into message dependence graph (MDG) that constituted thread dependence graph (TDG), which further comprised method dependence graph. For synchronization dependency in a concurrent program, the relationships like wait-notify and stop-join were taken into account. Zhang Guangquan *et al.* [126] used *LTL property* to extract a slicing criterion that reduced the count of states in a state transition graph. This approach eliminates the irrelevant statements from a concurrent program by using variable cache table (VCT) in conjunction with the *two pass slicing algorithm*.

For a program communicating via shared variables and interference dependence, Dennis Giffhorn *et al.* [127] extended the system dependence graph (SDG) to concurrent system dependence graph (cSDG). This work has evaluated the Nanda's and Krinke's algorithm for precise slicing and concluded that the algorithms do not scale well for larger programs or higher number of threads. Researchers [128] used a monitor-style model of concurrency and provided a more optimized and conservative algorithm as compared to that of Nanda's and Krinke's. As compared to the approaches presented earlier [124], [125], [128] that provided a static slice for concurrent programs, D.P. Mohapatra *et al.* [129] proposed an algorithm that uses *marking and unmarking* of edges in the CPDG, for computing the dynamic slices, without maintaining any execution trace. Researchers [130] proposed a dynamic slicing technique for the concurrent object-oriented program that uses the CSDG as program representation. J.T. Lallchandani *et al.* [131] also introduced a graph-based method that includes the marking and unmarking of dependence edge without using any trace file. In the proposed technique, concurrent CFG and PDG were developed to represent Java programs; and later the concurrent CFG was converted into object-oriented concurrent program dependence graph (OOCPDG) from where the dynamic slices were obtained by applying marking and unmarking for different dependencies.

To compute the dynamic slices in a concurrent program, a framework having *graph reachability* was presented by D. Goswami *et al.* [132] that handled the shared memory and message passing constructs. For each occurrence of the statement in a concurrent program with loops, the creation of a separate node in the execution trace has resulted into a precise slice. Venkatesh Prasad Ranganath *et al.* [133] presented *Escape analysis*, which reduced the count of ready dependence, and interference dependence edges in slicing concurrent Java programs. In the proposed technique, scenarios, where run time heap objects were reachable from a single thread, were detected and used further to prune the spurious interference dependence edges.

For static slicing of concurrent programs, an extended version of a *parallel algorithm* was presented by Diganta Goswami *et al.* [134]. Further, a parallel algorithm for static slicing of a concurrent program [135] was proposed, where a program initially got converted into a control flow graph and further into a concurrent control flow graph, on which slicing algorithm was applied.

The first publicly available program slicer for Java program was given by Ganeshan Jayaraman *et al.* [136] in which a library of classes, enabled an inter-procedural program slicer to slice the concurrent programs, without generating the SDG. Matthew B. Dwyer *et al.* [87] proposed a full *model reduction* technique for the model-based checking of a concurrent object oriented program. The first Non-SDG based slicing algorithm for a concurrent program was given by Venkatesh Prasad Ranganath *et al.* [137] in their seminal research paper. Researchers have presented the Indus framework, for analyzing and slicing a concurrent Java program in conjunction with an Eclipse based graphical user interface (GUI) named Kaveri.

John Hatcliff *et al.* [138] provided the *operational semantics* for a multithreaded language with concurrency primitives, and a bi-simulation based notion for the correctness of slice in multithreaded programs with infinite execution traces. Xiaofang Qi *et al.* [139] extended the concept of serialization to the parallel data flow, using thread interaction reachability graph (TIRG) and M-S pair program dependence graph (MSDG). In this approach, global *data dependence analysis* and a slicing algorithm using MSDG (M-S pair program dependence graph) were used for finding the precise program slice.

In the year 2003, Jens Krinke [140] presented the first slicing algorithm with *context sensitivity*, which accurately handled the slicing in recursive concurrent programs. All the previously available techniques for slicing concurrent program rely on in-lining of the called procedure, whereas the technique presented in this work captured the calling context through call string. Using GNU programs (named ctags, patch, diff) and benchmark database of the PROLANGS analysis framework (named gnugo, ansitape, assembler, cdecl, simulator, rolo, compiler, football, agrep, bison, flex), the same author [141], [142] evaluated the algorithms for slicing and chopping, and provided a solution to the problems related to the slice size. Krinke has also contributed in the field related to fine grained PDG and high precision approach to slice concurrent programs, *etc*.

For the first time, concurrent programs with nested loops and nested threads were taken into account by Mangla Gori Nanda *et al.* [143]. For slicing the shared memory concurrent programs, interleaving semantics and mutual exclusion were discussed. For finding the slice, CFG, threaded program dependence graph (tPDG) and *trace witness* were used. Jianjun Zaho *et al.* [144] computed the executable slices for concurrent program logic. The computations for the executable slices include the usage of argument dependence net, and an algorithm for calculating the sharing, communication and unification dependencies. Based on argument dependence net (AND), the slices for a concurrent program were calculated at the argument level that was considered more precise than a literal level. For dynamic slicing of the concurrent programs, Jingde Cheng [145] proposed two basic criteria, *viz.* completeness and soundness. It was shown that to obtain the complete and sound dynamic slices in a concurrent program, a dependence analysis method should be developed by self-measurement principle. For concurrent programs, the program slicing using incremental under-approximation technique was presented by Neha Rungta *et al.* [146]. For getting backward dynamic slices for Java byte-code program, a method that builds an instrumented JVM was presented by Attila Szegedi *et al.* [147]. A slice is said to be optimal if, under the abstraction of interpreting conditional branching as non-deterministic, it determines the statement minimal slice. Markus Muller-Olm *et al.* [148] commented that the optimal slicing of multithreaded programs with procedures and synchronizations was un-decidable. Sriraman Tallam *et al.* [149] presented an extended form of dynamic slicing for a multithreaded program, which assisted in locating faults that were caused by the data races. Here, dynamic slices represented the backward transitive closure over

exercised read after write data dependency and control dependency. This was shown that the dynamic slices of the multithreaded program must also contain inter thread write-after-read (WAR) and write-after-write (WAW) dependencies to capture the data race bugs efficiently. For finding executable slice and dependencies using Ada semantic interfaces specifications (ASIS) graph, Pierre Rousseau [150] presented a technique in which automatic program analysis tool named QUASAR was used with YASNOST as program slicer. Theoretical foundations for slicing a threaded program having Java monitors and wait/notify synchronization were described by Matthew B. Dwyer *et al.* [151]. This work has served as an extension over the previous work related to the extraction of slicing criteria from the temporal logic formulae that handled a multithreaded program with JVM-style concurrency primitives. Jens Krinke [152] proposed an algorithm that provided a more precise *static slice* in conjunction with interference handling for intra-procedural slicing. Proposed algorithm was for computing the static slice in which threaded CFG and threaded PDG both were used as an intermediate graphical representation. Zhenqiang Chen *et al.* [153] proposed a *Static slicing algorithm* that used CCFG and CPDG as an intermediate graphical representation for monitors-based concurrency model.

2.1.6 Formal method based testing

Formal methods are mathematical techniques used for the specification, development, and verification of software and hardware system. There are various techniques and algorithms proposed to test concurrent programs using formal methods, *viz.* Input Output Conformance (IOCO) test derivation, bakery algorithm, model checking, mechanical theorem proving, dynamical semantics, *etc*.

The formal theory for a concurrent program was developed by Stewart N.Weiss [154]. Based on the formal specifications, Jan Tretmans [155] presented a framework for testing a concurrent program in which *labeled transition systems* that provided the formal definitions of conformance, test execution, and test derivation were detailed.

Tevfik Bultan [156] presented a symbolic model checker to analyze the programs with unbounded integer domains for safety and liveness. The technique proposed by the researchers constituted the following concepts:

1. Constraints on integers, logical connectives and quantifiers were used to encode the transition relations and set of states symbolically.
2. Omega library was used for an efficient manipulation of the formulas to derive truth sets of temporal logic formulas, and their fix point computations.
3. Infinite-state programs were analyzed through conservative approximation techniques.

Parallel programs with high numerical computations were checked for their deterministic nature by Stephen F. Siegel *et al.* [157], where the researchers proposed a sequential model of a parallel program as final specifications. Deterministic nature was conformed by the equivalent output of both the *sequential and parallel model*. A limitation was applied on the count of loop iteration for handling the problem of state space explosion. A combination of formal and model-based methodology for testing the concurrent workflows was given by Chen-Wei Wang *et al.* [158], where a method sequence and a boolean guard condition were used to represent a test case.

Using general-purpose *mechanical theorem proving*, the framework for the specification and verification of reactive concurrent programs was provided by Sandip Ray *et al.* [159]. Specifications for concurrent programs were provided by formalizing the notion of refinements, which were analogous to stuttering trace containment. V. A. Vasenin *et al.* [160] used *Dynamical semantics* and presented a formal model for dynamical concurrent execution of programs. The model proposed by the researchers was used to guarantee the correctness of concurrent execution.

2.1.7 Random testing

Random testing (also called fuzz testing) is a technique where programs are tested by generating random and independent inputs, and the output is compared against the software specifications. There are various techniques and algorithms proposed to test concurrent program under random testing, *viz.* random partial order sampling, random subset selection algorithm, *etc*.

Koushik Sen [161] proposed a systematic algorithm for random testing of a concurrent program where threads were chosen at random schedule, and *partial ordered sampling* was used for removing the non-uniformity from the sample state space. Later in this field [162] *race directed random testing* was proposed that utilized potential data race information to separate real races from false races. A systematic review of the literature provides that for testing a concurrent program, random testing is an unexplored field and can be enhanced further.

2.1.8 Search-based testing

Search-based testing includes an application of meta-heuristic search techniques to the problem of test data generation. There are various classes like coverage matrices, search-based concurrency testing with and without noise injection, *etc.* in which search-based testing approach can be broadly classified.

Zdenek Letko [163] proposed an approach that used dynamic analysis to detect a concurrency bug along with a given execution path. Here, search-based techniques were used to control the noise generator to maximize the count of different interleavings, where *noise injection* was used to influence the thread interleaving. Behavioral aspects of the concurrent programs were identified for the existence of synchronization related errors. Using the IBM's concurrency testing infrastructure named ConTest, Bohuslav Krena *et al.* [164] described a search-based testing environment for testing the concurrent programs. Further, for the search-based testing of concurrent programs, coverage metric based on the algorithms like eraser, goldilocks, avio, goodlock, and happen-before has been proposed [165].

Table 2.2 highlights the major testing approaches used for testing concurrent programs, and the algorithms/techniques available under the specific approach to test a concurrent program.

Table 2.2: Algorithms/Techniques available under different approaches for testing concurrent program

S.No.	***Testing approach***	***Algorithm(s)/Technique(s) Available***	***Citation***
1.	Reachability testing	Replay-based approach	[29]
		Analysis algorithm	[30]
		Language-based approach	[31], [32]
		Race variant computation	[28], [33], [34]
		Deterministic testing	[35]
		Apportioning technique	[37]
		Race table construction	[38], [40]
		Reachability testing algorithm	[34], [38], [39], [40], [41], [42]
		Combination reduction	[43]
		Little strong happen before algorithm (LSHB)	[44]
		Using SYN sequence	[45], [46]
		Constructing ESYN-sequence	[48]
		Stateful reachability testing with variant and state pruning	[49]
		Partial order graph generation and event compression	[53]
2.	Structural testing	Coverage-based	[54], [166]
		Algorithm *Accept* and recursive function *propagate*	[55]
		Path analysis	[56], [57], [58], [59]
		Combination algorithm	[58]
		Path finding	[60], [61]
		Constraint-based	[58], [62]
		LTS	[63], [64]
		Concurrent state graph	[66]
		Post-mortem method	[67]
		Hot spot prioritization &	[68]

S.No.	*Testing approach*	*Algorithm(s)/Technique(s) Available*	*Citation*
		topological sort	
		Dataflow framework	[69]
		ACBP	[70]
		Tree-based algorithm	[75]
		Online Profiling Algorithm	[77]
3.	Model based testing	Flow back analysis and incremental tracing	[78]
		Branch and bound techniques	[79]
		Path generation algorithm, Co-path generation algorithm	[80]
		Test and Debugging algorithm	[81]
		Race and inconsistency detection	[82]
		Algorithm for generating sub event graph	[83]
		Change identification and test selection	[84]
		Value-schedule-based testing	[85]
		CQS	[15]
		Temporal sequencing	[86]
		Slicing-based	[86], [87], [88]
		Dependence analysis	[89]
		UML state machine	[90]
		ExitBlock algorithm	[92]
		Predicate coverage and use case sequence algorithm	[93]
		Message sequence path with BFS and DFS	[10]
		Labeled event structure and Formal specifications	[94]
		Bottleneck detection approach	[95]

S.No.	*Testing approach*	*Algorithm(s)/Technique(s) Available*	*Citation*
		Object-Z	[96]
		LTS	[97]
		SOFL specifications	[98]
		Transformation-based and Scenario oriented test case generation	[99]
		Trace visualization	[100], [101]
		Genetic algorithm	[105], [167]
4.	Mutation based	Mutant coverage algorithm	[106]
		Concurrency failure detection	[107]
		Fault based testing	[108]
		Binary set algorithm	[109]
		Mutation analysis	[110]
		Deterministic execution and mutation-based	[111]
		Using HAZOP	[112]
		Pseudo-code algorithms	[120]
		Selective Mutation	[121]
5.	Slicing based	Vertex reachability	[122], [123]
		Two pass slicing algorithm	[124], [125], [126]
		Two phase slicer	[127], [128]
		Marking and unmarking based	[129], [130], [131]
		Graph reachability	[132]
		Escape analysis and race detection	[133]
		Parallel algorithm	[134], [135]
		Model reduction	[87]
		Operational semantics	[138]
		Data dependence analysis	[139]
		Context sensitive approach	[140]
		Using trace witness	[143]

S.No.	*Testing approach*	*Algorithm(s)/Technique(s) Available*	*Citation*
6.	Formal methods	Labeled transition system	[155]
		Symbolic model	[156]
		Model checking	[157], [158]
		Mechanical theorem proving	[159]
		Dynamical semantics	[160]
7.	Random testing	Partial ordered sampling	[161]
		Race directed random testing	[162]
8.	Search based testing	Noise injection	[163], [164]
		Coverage matrices	[165]

2.2 Depictions used for testing the concurrent programs

Successful use of graphical elements in computer-related endeavors such as human interfaces and scientific visualization provide credible evidence of the extraordinary potential of visual communication. Program visualization deals with graphical presentation, monitoring, and exploration of programs expressed in a textual form. Graphical representation provides a view for a particular program that is independent of any programming language. Suitable graphical representation has been applied to the concurrent program (code or model) to get a midway representation, which further serves as an input for a specific technique used for testing the concurrent programs.

Table 2.3 represents the various categories of graphical representations used for concurrent programs. The main category for the graph is further subdivided into different subtypes where all the entries in subtype share the similar traits. Literature review indicates that flow graphs, dependence graphs, and UML diagrams are the most frequently used intermediate representations.

Table 2.3: Graphical representations used for testing a concurrent program.

S.No.	*Graphical representation*	*Diagrams included/Specific terminology used*	*Citation*
1	**Flow graph**	Control flow graph	[37], [55], [62], [85], [120], [138], [148], [151]

S.No.	*Graphical representation*	*Diagrams included/Specific terminology used*	*Citation*
		Program flow graph	[56], [57], [69], [138]
		Parallel program flow graph	[60], [61], [67], [69]
		Parallel control flow graph	[42], [59], [67], [73], [74], [166]
		Concurrent CFG	[93], [95], [130], [134], [135], [153]
		Threaded control flow graph	[143], [152]
		Extended concurrent CFG	[58], [84]
		Concurrent module flow graph	[70], [76]
		Conflict graph	[113]
		Java multithreaded flow diagram	[48]
		Concurrent control flow paths	[95]
		Event graph	[83]
		Event interaction graph	[80], [168]
		Conditional data flow diagram	[98]
		Thread call graph	[49]
		Symbolic transition graph	[86]
		D-graph	[66]
		Transaction graph	[169]
		Concurrent composite graph	[10]
		Exploration graph	[120]
		Race graph	[53]
		Occurrence nets	[79]

S.No.	*Graphical representation*	*Diagrams included/Specific terminology used*	*Citation*
		Control flow net	[122]
		Transition system model	[170]
		Labeled transition system	[41], [64], [108], [155]
		Annotated LTS	[63]
		Reachability graph	[37], [68]
		Thread interaction reachability graph	[139]
2	**Dependence graph**	Thread DG	[125], [149]
		Multithreaded DG	[124], [125],[126]
		Threaded interprocedural DG	[140]
		And-Or DG	[88]
		Use case DG	[93]
		Concurrent PDG	[129], [153]
		Concurrent SDG	[127], [128], [130]
		Threaded program dependence graph	[143], [152]
		Control dependence subgraph	[55]
		Access dependence graph	[101]
		Interprocedural PDG	[140], [141]
		OO Concurrent PDG	[131]
		Distributed PDG	[171]
		Static & Dynamic PDG	[78], [132]
		Dynamic dependence graph	[149], [172]
		Concurrency graph	[54], [132]
		Task graph	[30], [81]
		System dependence net	[123]
		Argument dependence net	[144]

S.No.	*Graphical representation*	*Diagrams included/Specific terminology used*	*Citation*
		Process dependence net	[122]
		Definition use net	[122]
		Process influence net	[122]
3	**UML diagram**	Activity diagram	[82], [15], [93], [99]
		Class diagram	[84], [90]
		Sequence diagram	[84], [15], [93], [10], [95], [100], [105]
		Multithreaded sequence diagram	[97]
		Collaboration diagram	[100]
		State machine model	[90]
		State chart diagram	[88]
		Use case diagram	[93]
4	**Space time diagram**	---	[34], [38], [81]
5	**Parse Tree**	---	[28]
6	**Object-Z model**	---	[96]
7	**Semantic interfaces specifications**	Ada semantic interfaces specifications	[150]
8	**PROMELA model**	---	[103]
9	**Rendezvous graph**	---	[56], [57]
10	**State space graph**	---	[116], [120]
11	**Resource allocation graph (RAG)**	---	[105]

2.3 Tools and Subject systems for testing concurrent programs

Based on the review of literature pertaining to the approaches and testing techniques available for testing a concurrent program, Table 2.4 lists the tools, toolset, prototype or APIs, available and the subject system (a component used for implementing or describing any technique or concept) or user program used under a particular testing approach for testing concurrent programs with their respective reference article and citation count.

Table 2.4: Tool/toolset/prototype/API and subject system/user program used under a particular testing approach

S.No.	*Testing approach*	*Tool/Toolset/Prototype/API*		*Subject System/program used*	
		Name	*Citation*	*Name*	*Citation*
1.	Reachability testing	TDCAda	[32], [33]	Bounded buffer	[32], [38], [39], [40], [47], [49], [51]
		Richtest	[38], [39], [40], [41]	Producer consumer	[28], [35], [37], [46], [48]
		Verisoft	[34], [38], [39], [40], [41], [47], [49]	Reader writer	[38], [39], [40], [41], [45], [47], [49]
		Inspect	[41], [49]	Dining philosopher	[38], [39], [40], [47]
		jCute	[41]	Ada program / package	[32]
		ValiMPI	[42]	User program – airline/crawler/FTPserver	[41]
		Exit block	[47]	Junit 3.8.1	[46]
		Java Path Finder II	[47]	Deterministic java replay utility	[46]
				Monitor-based program	[29]
				Message passing program	[33],[42]
				Use program-Send receive sequence	[34], [50]
2.	Structural testing	CATS	[54]	Bounder buffer	[57]
		ConTest	[56], [70]	Producer consumer	[57], [67], [73]
		Della pasta	[61]	Dining philosopher	[54]
		TCgen	[168]	Ada program/package	[168]
		ValiBPEL	[166]	GCD program	[59], [73], [166]
		ValiMPI	[59]	User program-Matrix multiplication	[73]
		METAFRAME/PAG	[62]	Jacobi	[59], [73]
		ValiPar	[73], [74]	Distributed alogorithm	[68]
		Maple	[77]	Path finding algorithm	[61]
				BPEL program	[58]
				User program-Send receive sequence	[71]
				TTE task tree	[75]
				User program-Mutex based	[70], [76]
				Mozilla data race bug	[72]
				User program-EXI	[56]
				Parallel program with parallel sections and events synchronization	[69]
				Case Study: Algorithm reduce, Counting int-T and int-L synchronizations, and Measuring the adequacy of the all int-T and all-int-L synchronizations coverage criteria	[63]
3.	Model based testing	Bandera java model checking	[87]	ATM system	[84], [95]
		CFD (Concurrency fault detector)	[167]	Bounded buffer	[87]
		ComTest (Comprehensive test tool)	[93]	Producer consumer	[90], [96], [100], [103]
		CWB-NC	[170]	Reader writer	[87]
		CCS	[170]	Dining philosopher	[167]
		In-situ partial order (ISP)	[104]	2 Thread mutex model	[97]
		Java path finder (JPF)	[87], [103]	Online shopping/store system	[82], [98]
		SPIN	[103]	Library information system	[93]
		LTSA tool	[97]	Gaussian elimination program	[91]
		SAL model checker	[90]	Conference protocol	[86]
		ConAn	[90]	UML diagram	[88], [10]

S.No.	Testing approach	Tool/Toolset/Prototype/API		Subject System/program used	
		Name	Citation	Name	Citation
		TCaseUML	[99]	Travel agency	[94]
		toc4j	[82]	Application-Repworker, raytracer, siena	[87]
		TorX tool	[86]	Distributed mutual exclusion circuit	[79]
		UML case tool	[101]	User program- With Java Multithreading	[80]
		VTune	[95]	Order processing system	[99]
		PerformaSure	[95]	Case study- Single server preemptive system	[83]
		TomcatProbe	[95]	Bellman-Ford iterative algorithm, 4 systolic diamond shaped computations, Divide and conquer, One-dimensional FFT, Gaussian elimination, A generic iteration, Master-salve, Master salave followed by Gaussian elimination, two partial differential equation iterative algorithms, A tree computation, A quantum dynamics algorithms, A tree computation, A quantum dynamics algorithm, Recursive Strassen algoithm for matrix multiplication, The Warshall algorithm for finding the transitive clouser of an adjacency matrix	[91]
		Jrat	[95]		
		NetBeans Profiler	[95]		
		JaDA	[100]		
		Jinsight	[100], [101]		
		JAVAVIS	[101]		
		TefKat	[90]		
		Bonus	[86]		
		TotX Tool Environment	[86]		
		jMoped	[87]		
		GAMBIT	[98]		
		mdb (Testing and debugging tool)	[81]		
		ANDES	[91]		
		Concurrency Fault Detector (CFD)	[105]		
4.	Mutation based	MuJava	[114], [116]	Bounded buffer	[111]
		MuTMuT	[116], [120]	Reader writer	[107]
		Jumble	[116]	User program – airline/crawler/FTPserver	[116]
		BugGen	[117]	GCD program	[118]
		Mothra	[117]	IBM concurrency benchmark	[115]
		ExMAn	[117]	Webserver	[108], [117]
		ConTest	[115]	TLM 2.0 library	[106]
		JPF	[115]	SystemC	[106]
		Bandera/Bogor	[115]	User program – Linked list	[116], [117]
		ConAn	[107], [115]	User program – With Java Multithreading	[109]
		Findbug	[107], [115]		
		Jlint	[107]		
		Comutation	[121]		
5.	Slicing based	Bandera tool set	[87], [138], [151]	Producer consumer	[124], [125], [126], [171]
		Concurrent Dynamic Slicer for Object Oriented Concurrent Programs	[131]	Compositional C++	[123]

S.No.	Testing approach	Tool/Toolset/Prototype/API		Subject System/program used	
		Name	Citation	Name	Citation
		(CDSOOCP)			
		Dynamic slicer for concurrent object oriented programs (DSCOP)	[130]	GNU programs	[141]
		Indus & Kaveri	[136], [137], [173]	PROLANGS analysis framework	[141]
		Slicing toolbox	[147]	Java card wallet	[127]
		VALSOFT	[142]	Java grande benchmark	[133]
		YASNOST	[150]	Occam 2 program	[122]
		SMV	[138], [151]	Mysql-1, 11	[149]
		SPIN	[138], [151]	Apache-1	[149]
		JPF	[87]	Splash-2 suite	[149]
		jMoped	[87]	User program - Matrix multiplication	[135]
		Bogor	[87]	Monitor based program	[153]
				Concurrent logic programming language (based on GHC)	[144]
				Common language runtime test case	[149]
				Two counter machine	[148]
6.	Formal methods	TVEDA	[155]	Producer consumer	[156]
		TGV	[155]	User program - Matrix multiplication	[157]
		TORX	[155]	Jacobi	[157]
		Omega library	[156]	Gaussian elimination program	[157]
		SPIN	[157]	Monte carlo algorithm	[157]
		SLAM toolkit	[157]	Hotel reservation system model	[158]
		MONA library	[156]		
7.	Random testing	CALFUZZER	[161]	User program – With Synchronized Collection Classes	[161]
8.	Search based testing	ConTest	[163], [165]	Dining philosopher	[165]
		Searchbestie	[164]	TIDOrbJ	[165]
				User program-airline/crawler/FTPserver	[164], [165]

2.4 Pros and Cons of approaches for testing concurrent programs

Based on the literature survey related to the approaches for testing the concurrent programs, Table 2.5 describes the pros and cons of different approaches adopted for testing the concurrent programs.

Table 2.5: Pros and Cons of various approaches adopted for testing the concurrent programs.

S. No.	Pros & Cons of Testing approaches
1.	***Reachability testing*** ***Pros*** • In combination with a suitable coverage criteria (*e.g.* path coverage, decision coverage, branch coverage, *etc.*), it improves the quality of testing for a

S. No.	*Pros & Cons of Testing approaches*
	concurrent program by selecting suitable synchronization events and avoiding the unnecessary synchronization sequences or test cases [42]. ***Cons*** • Involvement of non-deterministic execution order of the events in a program may lead to the sequence explosion problem, which poses a challenge in generating a good set of SYN-sequences. • This is an implementation-based approach, where test sequences are generated on-the-fly and without constructing any static model. Therefore the approaches based on the model-based fault detection cannot be applied, which may save the resources (later in SDLC phases) by discarding the faulty sequences at the early phase of development. • This is an exhaustive approach, where all the possible SYN-sequences are intended to execute on the subject system. Exhaustive testing can maximize the test coverage, but it is often impractical to accomplish the task due to resource related constraints (like memory space need to save the sequences, time utilized to execute the whole set of SYN-sequences) [174].
2.	***Structural testing*** ***Pros*** • High fault coverage can be achieved by automating the process of test pattern generation in structural testing. • It may assist in code optimization by spotting the dead code in a concurrent program. ***Cons*** • For a concurrent program, non-deterministic execution order makes it tedious to determine the minimum set of test cases required to complete the coverage.
3.	***Model-based testing*** ***Pros*** • Verification of the execution with the expected behavior can be performed using the models as test oracles [175]. • Creation of formal test models at the initial stage of SDLC helps in finding the faults and inconsistencies within the requirement specifications [176]. ***Cons***

S. No.	*Pros & Cons of Testing approaches*
	• It might be more difficult and time-consuming to find the cause of the failed test because when any of the generated tests fails, it must be decided that whether the failure has been caused by the software under test (SUT), the adaptor code, or an error in the model [177]. • During SDLC if there be any change in the requirements, then it will make the previous model outdated and test cases as generated using the earlier model will not be of any use. • It demands an extra effort regarding having few additional skills from the testers, which include knowledge of different forms of state machines, formal languages, and automata theory [178].
4.	***Mutation-based*** ***Pros*** • It's flexible to apply at different levels in testing using different languages, object-oriented programs and formal specifications [110]. • One can guarantee for the absence of some particular type of errors from the original program by killing all non-equivalent mutants corresponding to that error [179]. • The tester needs to develop a test data that is capable of killing all the generated mutants. Hence, one can generate an effective test data set, which is powerful enough to find errors in the program. ***Cons*** • It is quite costly in terms of time and effort involved. Even for a small program, a large number of mutants are generated and needed to be run for comparing the outcomes with the original, un-mutated program [179]. • Mutants those are functionally equivalent to the original program need manual interference for distinguishing, when compared with the original programs. For example, if a variable is never zero at a decision in which it is compared with zero using the "<" operator, the mutant in which that operator is replaced with "<=" operator will be equivalent.
5.	***Slicing-based*** ***Pros*** • It visualizes dependencies and restricts the attention to only the components

S. No.	*Pros & Cons of Testing approaches*
	of a program, which are relevant to the evaluation of certain expressions. • It helps in program comprehension, maintenance, testing and debugging, comparison of program versions and reuse of the code. ***Cons*** • Computation of dynamic slices is based on a particular program execution, which incurs a high run-time overhead due to the recording of the program execution and/or run-time analysis of every executed statement [180].
6.	***Formal methods*** ***Pros*** • It helps in automation of the test case derivation from the formal specification [155]. • There are some properties of the system that can only be verified and not validated. Formal methods open ways to combine the verification and testing systematically by providing a more precise and unambiguous way for describing the specifications [155]. ***Cons*** • It requires an extensive labor to axiomatize and solve the verification problems [181].
7.	***Random testing*** ***Pros*** • Generation of random test for the software can infer the quantitative estimates related to its operational reliability. ***Cons*** • Due to its selective nature, it provides moderate coverage for the module under test and hence it is ineffective for the random-pattern-resistant faults.
8.	***Search-based testing*** ***Pros*** • It can be easily adapted to new testing objectives by redefining the fitness function and optimize the tests towards different functional and non-functional properties like coverage criteria. • These techniques extensively involve the usage of evolutionary algorithms, which may lead to high code coverage while test case generation. ***Cons***

S. No.	*Pros & Cons of Testing approaches*
	• For a large search space with no diversified population, usage of SBT may result into an inefficient outcome as it depends upon the size of search space, the diversity of their initial population, and the effectiveness of their fitness functions.

2.5 Research Gaps/Inferences

This literature review, undertaken in this chapter, focuses on identifying the research gaps in the area of testing concurrent programs. Area related to the testing concurrent program has been divided into *eight* categories, and the implications related to each testing approaches have been discussed in the following subsections.

2.5.1 Reachability testing

Area related to an efficient organization of partially-ordered event sequences can be explored further by applying multiplexing of partially-ordered events. Research in the area of automatic generation of test drivers from the test sequences can be extended by using incremental causal reasoning. Greedoid set system can be applied to the algorithm that can selectively execute a set of test sequences according to some coverage criteria. Matroid theory under greedy algorithms may enhance the fault revelation in concurrent programs when applied to the combination of coverage criteria and reachability testing.

2.5.2 Structural testing

Methods like ample set, stubborn set, and persistent set/sleep set algorithms under partial order reduction techniques can be applied to reduce the size of a test suite for the concurrent programs. A more restrictive partial order using symmetric commutative forward and backward relations may be used for the systematic evaluation of program-based coverage criteria for the concurrent programs. For testing parallel programs, the structural testing criteria like all-nodes-s (related to send nodes) criterion, all-nodes-r (related to receive nodes) criterion, and all-s-uses (related to send nodes) criterion can be explored to increase the effectiveness of fault detection. Simultaneous reachability analysis (SRA) may be applied to incremental integration testing of concurrent programs. SRA can be used with compositional techniques, where compositional

technique builds reachability graphs in a modular fashion and reducing the graph before using them for composing the larger ones.

2.5.3 Model-based testing

The technique named sub-tree mining can be applied to sub-event graphs to find its optimized count. Visualization of the dormancy problem in a concurrent program can be achieved through colored Petri-nets in an efficient way. Research in the area of program size reduction before model checking can be performed by applying amorphous slicing and semantic cloning to a concurrent program. Test coverage can be enhanced by removing cloned code, by executing more sophisticated tests (like data access coverage and data range coverage), and by inserting fault injections in the program source code. Fault detection effectiveness can be enhanced by using layered program auralization into a statement-coverage test data adequacy criterion. A reduced partial order-based model, called Mazurkiewicz Trace Machine (MTM), may be used for efficient test case generation for distributed systems.

2.5.4 Mutation testing

For distributed systems, mutation testing in conjunction with particle swarm optimization (PSO) may effectively explore the faults like deadlock. To find the set of mutants, a concurrent program can be enhanced by using a population-based search algorithm like bees algorithm. Combinatorial optimization may improve the coverage criteria for testing a concurrent program.

2.5.5 Slicing-based testing

More enhanced points-to analysis algorithm like hybrid context sensitive may be applied to slicing algorithms for better precision of the slice. Ant colony optimization and swarm intelligence may improve the slice and its precision for a concurrent program. For improving the process of architectural level slicing, the formal architectural description language may be used to specify software architecture more formally. Using dynamic software architecture slicing, an efficient dependence analysis may be performed between the components at system architecture level. Research in the area of architectural dependence analysis may be improved by incorporating C2-style architecture testing with dependence type hierarchy.

2.5.6 Formal method-based testing

For better state partitioning in symbolic model checking of the concurrent programs, application of invariant-based compositional verification rule may provide better results. Under model checking, symbolic algebra, and first-order theorem under automated theorem proving can be used to improve the verification of parallel numerical programs. For deciding the feasibility of inter-leavings in a concurrent program, further research may be undertaken by using escape analysis and start-join analysis as an improvement over block-based algorithm.

2.5.7 Random testing

For random testing of a concurrent program, random partial order sampling algorithm in conjunction with genetic algorithms - partial least squares (GA-PLS), and particle swarm optimization (PSO) may provide better results regarding partial order sample.

2.5.8 Search-based testing

Better heuristic search techniques like Greedy search, Iterative Deepening A*, Simplified Memory-bounded A* may be used to maximize the number of different interleaving in concurrent programs. A combination of search-based and constraint-based testing may be used to achieve higher coverage for generating the test data for testing the concurrent programs.

After going through the process of rigorous observations, extensive overviews, detailed literature survey and inferences from the gap analysis of the approaches related to testing concurrent programs, it can be formulated that there exists an opulent space for combining the utility of meta-heuristic approaches to model-based testing to explore the structural aspects in the UML model by precise application of the heuristics in context of fetching the slice of feasible test scenarios from the set having feasible as well as in-feasible ones.

2.6 Heuristics and Meta-heuristic

The word ‘heuristic’ has been derived from the Greek word ‘εὑρίσκω’ which means ‘to find’; and it has been given to the basic form of approximate algorithms. In order to get near-optimum solutions in reasonable and practical computational times, the

approximate algorithms sacrificed the guarantee of finding an optimal solution. Meta-heuristic combines heuristic methods in higher level frameworks and aims for an efficient and effective exploration of the search space driven by the protocols avoiding the trap of local optimum. As compared to heuristic, the meta-heuristics need special information about the problem to be solved to obtain good results. Hence, these are usually more difficult to implement and tune. Meta-heuristic approaches rely on probabilistic decisions made during the search. The feature that segregates it from pure random search is that meta-heuristics use randomness in an intelligent and biased form. Where, biasness is based on the objective function, previously made decisions or on prior performance [182].

Meta-heuristic is faster as compared to an exhaustive search in the context of computational effort. Meta-heuristics use stochastic behavior for searching operation for altering one or more initial candidate solutions as generated by random sampling process in the search space. For dealing with the inherent complexities of real world problems and considering the importance of classical/exact approaches in the domain related to system development or research, the developers and the researchers looked for meta-heuristic approaches for getting a near-optimal solution in a computationally tractable manner. Otherwise, there will be a wait for developing a provable approach to solving the existing problems. In recent past years, the meta-heuristics find an appreciable rate of acceptance due to its ability to handle a variety of complexities associated with real world problems and providing a logically acceptable solution [183].

The hardness related to any problem indicates the complexity related to the problem under consideration. For polynomial instances, where the power of the polynomial function represents the large algorithmic complexity, the real life instances cannot be evaluated in an affordable time span (*e.g.*, a complexity of $O(n^{5000})$). Instead of the problem complexity, another vital parameter that demands to be considered is the search time. For real time search constraints in the polynomial problem domain, where state-of-art exact approaches cannot handle size and structure within the required search time, the usage of meta-heuristic approaches is quite justified. For NP class of problem domains, the exact algorithms need exponential time. The time required for solving a designated problem may vary from few seconds to few months (production versus design problems) [184].

Most meta-heuristic methods are motivated by natural, physical or biological principles and try to mimic them at a fundamental level through various operators. A common theme seen across all meta-heuristics is the balance between exploration and exploitation. Exploration refers to how well the operators diversify solutions in the search space. This aspect gives the meta-heuristic a global search behavior. Exploitation refers to how well the operators can utilize the information available from solutions from previous iterations to intensify search. Such intensification provides the meta-heuristic a local search characteristic. Some meta-heuristics tend to be more explorative than exploitative, while some others do the opposite. For example, the primitive method of randomly picking solutions for a certain number of iterations represents a completely explorative search. On the other hand, hill climbing where the current solution is incrementally modified until it improves is an example of a completely exploitative search. More commonly, meta-heuristics allow this balance between diversification and intensification to be adjusted by the user through operator parameters [183], [184].

2.7 Sources of inducement

Nature has provided a good source of inspiration for various algorithms related to the domain of optimization. There exist various origins from physics, chemistry, and biology, which are further referred to as nature-inspired. Mostly, nature-inspired algorithms follow the affluent traits of the existing biological system. Hence, a large subset of nature-inspired approaches is referred to as bio-inspired. Swarm intelligence is a specific class of the bio-inspired algorithms. Bio-inspired approaches are further classified by their inspiration sources as follows [184], [185]:

2.7.1 Swarm intelligence-based approaches

Swarm intelligence (SI) refers to the protocols that include the behavior of multiple, interacting agents following the simple rules of nature collectively. In a group of living organisms, a solo entity may be taken as unintelligent, but the complete system with multiple agents showing self-organization demonstrates a cooperative intellect. The prime traits followed by the SI-based approaches are summarized as hereunder:

- Information sharing among the agents present in the system.
- Self-management and coordinated evolution.
- Learning with coordination.

- Easy to port for solving the real-time concerns.
- Simple to parallelize for practical usage.

2.7.2 Bio-inspired, but not employing swarm intelligence

As is evident from the above section, SI majorly belongs to a set of bio-inspired algorithms. It's inferential regarding categorization that the SI-based ⊂ bio-inspired ⊂nature-inspired. Methodologies like Genetic algorithms (GA), which follow the evolutionary approach, are well-defined bio-inspired but do not come under the category of SI-based as there exists no cooperative behavior for the elements/entities present under the population set. One seminal example in this same category is Xin-She Yang's [183] flower pollination algorithm, which comes under bio-inspired, but it doesn't belong to SI. This approach mimics the pollination process of the flowering plants as executed by the pollinating insects.

2.7.3 Motivation from physics and/or chemistry

Phenomena that exist in physics and chemistry also provide a push for drafting some meta-heuristic approaches, where the occurrence of events mimics certain physical or chemical principles, including gravity, electrical charge, river system, *etc.* Various approaches that belong to the said category are water cycle algorithm, harmony search, spiral optimization, and simulated annealing, *etc.*

2.8 Design space of meta-heuristic

As shown in Figure 2.2, while drafting for a meta-heuristic, two contrasting measures that are needed to be taken care of are diversification and intensification, where former is related to search space exploration, and later one implies exploration of the best solution found. Best solutions among a set of solutions are determined by the promising regions as obtained from the said two approaches. Intensification explores the promising areas more exhaustively for finding better solutions. Whereas, in diversification the search is not restricted to a limited number of regions; and hence non-explored regions are traversed for ensuring the even exploration of the whole search space. In terms of exploration, the random search and local search are the extreme search approaches corresponding to exploration and exploitation. Random search results into the generation of a random solution in the search space for every iteration without consuming any

memory units. For the improvement of current solution in the basic local search approach, one selects the best solution from the neighborhood at each iteration [185].

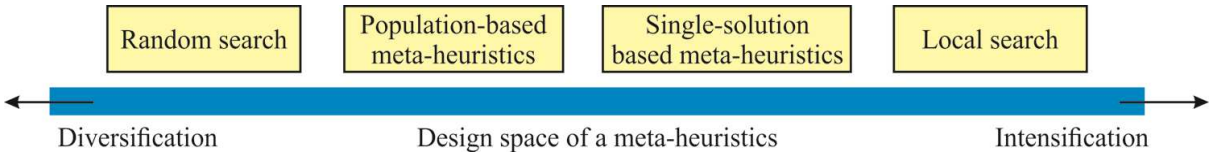


Figure 2.2: Criteria for designing a meta-heuristic.

2.8.1 Characteristics of a meta-heuristic

Blum and Roli [186] sketched certain properties related to meta-heuristics, which are as follows:

- Meta-heuristics are the policies, which lead the process of searching.
- Ultimate aim is to inspect the search space for finding the optimal or near-optimal solutions.
- It's diversified from easy local search procedures to complicated learning processes.
- These approaches are generally non-deterministic and provide approximate solutions.
- It may have the machinery for avoiding the tarp in limited regions of search space.
- Generally, meta-heuristics are independent of any specific problem area, *i.e.*, one meta-heuristic technique may be applied for diversified domains in real life. For example, ant colony optimization can be applied for traveling salesman problem, vehicle routing problem, pavement designing problem, pressure valve design problem, *etc*.

2.8.2 Problem domains of a meta-heuristic

In realistic scenarios, traditional algorithms may not perform appropriately for some specific set of problem domains; and hence, the computational intelligent methodologies taking inspiration from nature are found suitable. The present research work is also the outcome of such methodologies. A particular concern is said to be difficult if that follows the below-given criteria as provided by Michalewicz and Fogel in their work [187]:

- The case when count for possible outcomes in the given search space is so huge to get the best solution and hence, prevents the search for being an exhaustive one.

- The scenario where evaluation function which explains the worth for the proposed solution varies *w.r.t.* time or it is noisy. And as a result, not just a single outcome, but an entire group of solutions appear.

- When complexity of the problem under consideration pushes to use a simplified illustration of the problem; and the resulted solution is useless at all.

- When potential outcomes are unbalanced for constructing even a single possible solution and hence, searching for an optimal one is strenuous.

2.8.3 Significance of using a meta-heuristic approach

Following are the various advantages of using meta-heuristic approaches over the traditional optimization methodologies [184]:

1. For computationally easy problems with complex and large input size (P class problem in the context of data-structures and algorithms), the hurdle as faced by the traditional methodologies to handle input data can be overcome by usage of meta-heuristics.
2. The problem domains where no known exact approach exists for solving the concern in a realistic time frame (NP-hard), the meta-heuristic may provide an adequate solution.
3. Meta-heuristic can be applied in conjunction with simulation-oriented, non-analytic or black-box objective functions, as it wouldn't require any gradient information.
4. Inherent stochasticity in most of the meta-heuristics provides an ability to get recovered from local optima and handling uncertainties in the objective function.
5. Multiple objectives can be handled with small changes in the same meta-heuristic algorithm.

2.8.4 Application areas of meta-heuristics

The domains with large-sized problem instances find the robust application of meta-heuristics for dispatching acceptable solutions in an admissible time frame. These approaches wouldn't guarantee for giving the solutions that are either global optimal or even bounded. Meta-heuristics can be applied in a large number of application areas. Some of these are as follows [185]:

- Computational biology in conjunction with machine learning and data mining
- Bioinformatics
- Image processing
- System modeling
- Structural optimization in civil and mechanical fields
- Topology optimization in electrical and electronics
- Robotics
- Scheduling and production problems
- Planning in routing problems
- Logistics and transportation
- Supply chain management.

2.9 A generic framework for a meta-heuristic

Almost all the meta-heuristics approaches follow a related pattern of operations and hence can be considered within a generic structure [185], [188] as shown in Algorithm 1.

Algorithm 1 A common framework for a meta-heuristic

Requirements: SP, GP, RP, UP, TP, $N(\geq 1), \mu(\leq N), \lambda, \rho(\leq N)$

1: Position an iteration counter as $t = 0$

2: Initialize N solutions randomly and place these in a set S_t

3: Start assessing every element of set S_t

4: X_t^* denotes the best solutions available from set S_t

5: **Loop**

6: Using a selection plan (SP) to choose μ solutions (set P_t) from S_t

7: A generation plan (GP) is used to create λ new solutions (set C_t) from P_t

8: ρ solutions (set R_t) will be chosen from S_t following a replacement plan (RP)

9: Using ρ solutions from a pool derived from P_t, C_t, and R_t and an update plan (UP), S_t will get updated by replacing R_t

10: Each element of S_t gets evaluated

11: X_t^* will get updated by identifying the best solution from S_t

12: Increment the counter, $t \leftarrow t + 1$

13: **Until** (satisfaction of the termination plan (TP))

14: X_t^* will provide the near-optimal solution

Initial *Requirements* show that for crafting a meta-heuristic methodology, a minimum of five planning constructs (SP, GP, RP, UP, TP) and four parameters (N, μ, λ, ρ) are required. Independent plan may indulge one or more parameters of their own.

Step 1 initiates the loop counter (t) by assigning a start value zero. The repeat-until construct gives an iterative nature to the generic framework of any meta-heuristic algorithm.

Step 2 produces an elementary set S_t with N solutions, which are created randomly within the prescribed lower and upper bounds. It creates random permutations of entities for the domain of combinatorial optimization. To permit exploration in a specific problem domain, the inclusion of good initial solutions enhances the worth of the set of selected population elements.

Step 3 estimates each (N) member of set S_t by using the given objective and constraint functions. To give an evaluation pattern, constraint violation, if any, must be taken into account for using in subsequent steps of the selected algorithm. One mode for defining constraint violation $CV(x/p)$ is specified as follows:

$$CV(x/p) = \sum_{j=1}^{J} \langle g_i(x/p) \rangle$$

Here, $\langle \alpha \rangle$ is $|\alpha|$ if $\alpha < 0$, otherwise it is considered as zero. Normalization must be performed prior adding the constraint values related to different constraints. For performing a satisfactory evaluation for the solutions, subsequent steps use the objective function value $f(x)$ and constraint violation $CV(x/p)$ value.

Step 4 computes X_t^* which determines the best member of the set S_t using pair-wise contrasting among set members. For constrained optimization problems, one among the following strategy may be implied to get one solution from the two existing solutions (A and B):

1. When both A and B are viable, then opt for the one with lesser objective (f) value,
2. Choose A, when A seems to be feasible and B is not and vice versa,
3. In case both the solutions are not feasible, then choose for the one with lesser value for the CV.

Step 5 pushes the technique into an iterative construct (lined from Step 6 to 12) until the plan for its termination gets satisfied in Step 13. TP is one among the five constructs as chosen by the user, which may include attainment of a pre-specified target objective value (f_T) that will be fulfilled when condition $f(X_t^*) \leq f_T$ becomes true. Various values for TP are quite possible, where it may involve a pre-set count for iterations T with termination condition as $t \geq T$.

Step 6 uses a selection plan (SP) for choosing μ solutions from the set S_t. Using f and CV, S_t has been analyzed by SP for selecting best solutions out of it. Based on SP used, there exist variants of any specific meta-heuristic approach. A new set P_t has been formed by μ solutions.

In ***Step 7***, a generation plan (GP) is used for P_t to produce a set of λ new solutions. This GP gives the main thrust for the search operation of the meta-heuristic approach. GP varies for combinatorial as well as functional optimization problem areas. Final solutions as achieved from this step come under the set C_t.

In ***Step 8***, a replacement plan (RP) works to replace the ρ worse or randomly chosen solutions from set S_t and further save it in R_t. For some meta-heuristic techniques, the set R_t can simply be taken as P_t (with condition $\rho = \mu$). For making the approach more greedy, R_t can be ρ worst members of S_t.

Step 9 replaces ρ selected solutions from S_t by ρ other solutions. Here, an update plan (UP) has been used for choosing ρ solutions from a pool of at most $(\mu + \lambda + \rho)$ solutions taken from the combined pool originated from $P_t \cup C_t \cup R_t$. In the case of $R_t = P_t$ the combined pool need not have the sets R_t and P_t for avoiding duplication of solutions. UP simply chooses the best ρ solutions from the pool, which may constitute $R_t \cup C_t$ or $P_t \cup C_t$ or simply C_t. Elite-preserving trait of a meta-heuristic implies the inclusion of members of R_t in the combined pool. For determining the robustness of any meta-heuristic approach, RP and UP play a vital role. The design of RPs and UPs complies with the diversification in the candidate solutions. Following greedy methodology and replacing worst solutions with the best solutions sometimes leads to faster but premature convergence.

Step 10 assesses every new member of S_t , while ***Step 11*** uses the best elements of updated S_t; and updates the set X_t^* which contains the best members. The current-best solution is stated as near-optimal after the algorithm has satisfied TP.

By setting the value of N=1, the above given meta-heuristic approach can be used for single solution optimization. This will also force the values for μ and ρ as $\mu = \rho = 1$. In Step 7, when $\lambda = 1$, it will create a single new solution. In this scenario, SP and RP are the simple methods to choose the singleton solution in S_t. Therefore, $P_t = R_t = S_t$. Gaussian or other distribution can be used by generation plan (GP) for choosing a solution in the neighborhood of singleton solution in S_t. An update plan (UP) implies a selection of a single solution either from $R_t \cup C_t$ (elite preservation) or opting for a single new solution from C_t alone and replacing P_t.

Various meta-heuristic algorithms are being developed by careful analysis of the above-said five plans and choosing their appropriate size/dimensions. Plans here involve the stochastic operations, thereby making the final approach stochastic.

2.10 Problem Formulation

In an activity diagram of a use case, a series of activities is known as test scenario or just a scenario. In the context of software development, testing and maintenance of object-oriented systems, every scenario is related to a trail of an execution of activities in the system. To test a use case adequately, it is necessary to recognize all feasible begin-to-end paths in an activity diagram for covering all activities with its associated control constructs. The control flow constructs, *viz.* decision, looping, synchronization, concurrency, *etc.* decide the execution sequence of various activities. A fork-join node in the UML activity diagram serves as a concurrent entity. The existence of a fork element commences several parallel control flows. The default activity related to join is "AND" that indicates when all the in-flows to a particular join node are complete, the final flow is directed on out-edge of a join element. The non-deterministic characteristic of concurrency allows the switching after any activity node of the sub-queues existing between fork-join. This interleaving result into permutation among activity nodes; and the same leads to a sequence explosion for test scenarios that are created amongst fork-join. Therefore, test scenario generation for a concurrent segment in the UML activity diagram is considered equivalent to the problem of combinatorial optimization.

The literature review conducted here encompasses a wide-ranging survey of well-researched papers and inferential observations for the homogeneous groupings, such as testing concurrent programs, concurrency in UML models, and model-based testing with concurrency studied to infer that the work related to test scenario (or path) generation is conducted by human intervention and mostly by applying the exact methods. The survey suggests that there lies the opulence of scope for using the meta-heuristic techniques for generating the test scenarios using the UML activity diagram. The reasonable use of the meta-heuristic algorithm is highly recommended to get the effective solution in cases where popularly practiced exact algorithms cannot efficiently control the instances (size, structure) ranging within the necessitated search time [185], [183]. In the recent decades, multiple choices of bio-inspired methods have been anticipated to generate test scenarios for the concurrent section in an activity diagram. All these works have either huge redundancy in generated test scenarios or have a high randomness that results into a less number of final test scenarios. This factor has driven us to explore the utility of meta-heuristic technique that involves less time with a better exploration of search space.

2.10.1 Scope of work

Example: Consider a sample activity diagram containing fork-join construct as shown in Figure 2.3. Here, we can see that there are two sub-queues, namely, $5 \rightarrow 6 \rightarrow 7$ and $8 \rightarrow 9$ between fork-join node.

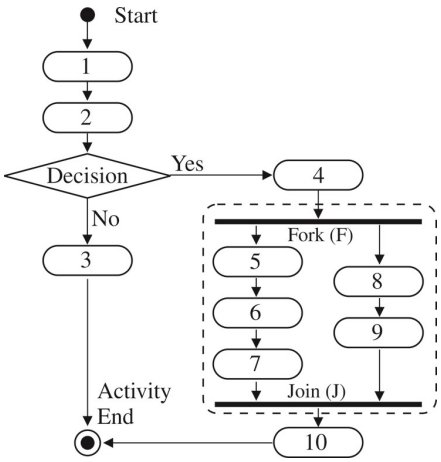


Figure 2.3: An activity diagram with concurrent element.

The incoming control flow creates the concurrent control flow on each out edge at fork node. The join node combines multiple inflows into a single flow. A single control flow is produced only when all the incoming control flows have been completed at the output node. In concurrency, multiple flows of activities start at fork node simultaneously; and the same might produce a permutation for next activity node(s) in test scenario (or sequence) by alternating control between nodes under fork-join. In the absence of switch point(s) in concurrent sub-queues, all nodes will get executed in the serial order, consequently ensuing loss of concurrency among the activities.

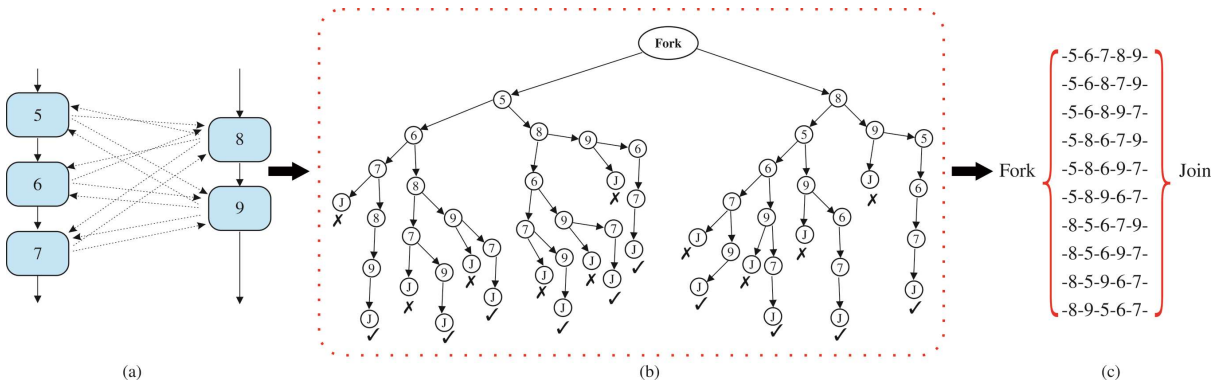


Figure 2.4: Concurrent region of activity diagram with an exploratory tree structure and final feasible scenarios.

Figure 2.4(a) shows the interleaved execution and the switching after each node present in the sub-queues, which result into an exponential sequence count for the generated test scenarios. The tree structure in Figure 2.4(b) depicts an exploratory view with all the valid and invalid scenarios generated. In the above diagram, the ✓ stands for a valid path; and × indicates an invalid path. Figure 2.4(c) represents the set of feasible test scenarios generated from the sub-queues, namely, 5 → 6 → 7 and 8→ 9.

The case becomes problematic when the count of sub-queues and (or) the node count in the sub-queues varies increasingly. Both the situations lead to enhance the number of permutations among the nodes and henceforth raise the count of test sequences exponentially. The combinatorial explosion with regard to the number of succeeding test scenarios by adopting the formula $\frac{(n_1+n_2+\cdots n_i)!}{(n_1!\times n_2!\times \ldots\ldots n_i!)}$ is described in Table 2.6. In the formula used, *i* stands for the count of sub-queues and *n* represents the number of activity nodes in i^{th} sub-queue [189].

Table 2.6: Combinatorial explosion of test scenarios for variable size and count of fork-join queues.

Case label	Count of fork-join sub-queues	Elements in each sub-queue	Count of feasible test scenarios generated using formula [189]
a.	2	$q_1 = 3, q_2 = 2$	10
b.	2	$q_1 = 5, q_2 = 4$	120
c.	3	$q_1 = 5, q_2 = 4, q_3 = 3$	27720
d.	4	$q_1 = 5, q_2 = 4, q_3 = 3, q_4 = 2$	2522520

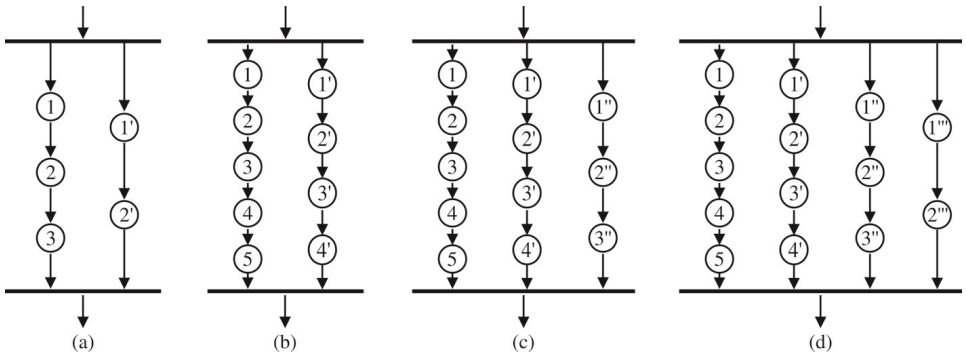


Figure 2.5: Variable count of activity nodes and sub-queues under fork-join.

Figure 2.5 presents four sample activity diagrams corresponding to the cases listed in Table 2.6. Table 2.7 shows that a wait time of more than 7 minutes exists while executing Depth-first Search (DFS) [190], [191] on Case d.

Table 2.7: Count of test scenarios (CTS) and algorithm execution time (AET) in millisecond obtained using DFS exact algorithm and meta-heuristic approaches on the synthetic activity diagram in Figure 2.5. (Population in GA or number of ants in ACO is taken as 100 and number of generations in GA or number of trials in ACO is taken as 10). Here, average results obtained after executing the specific approach with 30 instance runs have been listed.

Activity diagram	**DFS** [191]		**ACO** [192]		**GA** [189]	
	CTS	*AET(ms)*	*CTS*	*AET(ms)*	*CTS*	*AET(ms)*
a	10	1.5	10	418	10	117
b	120	13	57	928	0	30
c	27720	1149	610	2412	0	26
d	No result after wait of 7 minutes		1424	4499	0	28

In the case of moderate and significant instances, it would be quite impractical to assess all possible solutions received through exact algorithms like DFS, Breadth-first Search, *etc.* [191] as it becomes a time-consuming process. Here, the application of ACO [192] is providing a good number of feasible test scenarios, but this count can be increased further by using such a variant where the agent is less redundant in following the same path repeatedly. As the path with high pheromone accumulation becomes influential, there exists duplicity in the traversed paths. For Case b, c and d, GA fails as the count of activity nodes in any sub-queue or number of sub-queues increases. GA (or EA) [189] uses a random function to generate and select the final scenarios. However, it remains inefficient in offering the desired number as well as quality due to the large set of vague test scenarios resulted from inherent randomness.

2.10.2 Meta-heuristics to generate scenario(s) from UML Activity diagram

In a software system, test scenarios can be produced by examining either design level activity diagram or its corresponding source code. UML-based scenario generation has the subsequent edge over code-based scenario generation. First, we require fewer quanta of data to be processed alternative to the large code. Second, we can discover test

scenarios at the early stage of system (or software) development life cycle [17]. The same enables software analysts, developers or testers to build a useful test plan and development pattern before the start of the development phase. In this section, we further study the existing research work on test scenario generation using meta-heuristic techniques. Table 2.8 highlights the contribution of those researchers who have used bio-inspired techniques for generating scenarios from UML activity diagrams.

Table 2.8: Related work for test scenario (or sequence) generation using meta-heuristic techniques.

S.No.	Year	Author(s)	Meta-heuristic used	Drawback
1.	2005	D. Xu, H. Li, and C.P. Lam [16]	Behavior of bacteria (Maxococcus xanthus) as agent	Biologically inspired agents always use some probability factor for its directional orientation (like pheromone in ant colony and flux in amoeboid approach). The study does not ponder much upon the fact that governs the movement of the agent in any specific direction. The agent will clone, as and when required. The exploration of scenario tree uses an approach similar to parallel graph search to obtain all the scenarios, which come under exact algorithm. For a large search space that exists for the problem here, exact algorithms are not viable. The study lacks any benchmark subject system to validate the proposed approach.
2.	2005 2010	H. Li and C. P. Lam [193] C. P. Lam	Agents using Anti-ant variant	There exists overstrain of converting an activity diagram to a state machine diagram first. The problem of scalability of the

S.No.	Year	Author(s)	Meta-heuristic used	Drawback
		[194]		proposed algorithm is not discussed here.
3.	2008	U. Farooq, C. P. Lam, and H. Li [195]	Random walk algorithm	Symmetric flows turn asymmetric due to the selection of the wrong node that may result in a noisy test sequence.
4.	2012	M. Shirole, M. Kommuri, and R. Kumar [189]	Evolutionary algorithm (EA) [where EA is a variant of GA]	Values for the various parameters like selection percentage, crossover percentage, and crossover point have not been provided. With an increase in the count of nodes in sub-queues under fork-join, the probability of getting the right sequence diminishes due to the cumulative effect of randomness at various steps in EA.
5.	2014	A. Mishra and D. P. Mohapatra [196]	Ant colony optimization	The interleaved execution of nodes under the concurrent section of activity diagram has not been discussed. ACO has been used for test sequence prioritization but not for their generation. The study has missed demonstration of the fork-join nodes in sample model taken for validating the proposed approach.
6.	2015	F. Sayyari and S. Emadi [197]	ACO with Markov chain	Researchers have missed giving the discussion on UML models having concurrency construct. Benchmark models are not taken to validate their proposed approach.

Observation: If we perceive research trend for test scenario (or sequence) generation using meta-heuristics, we can observe that the earlier researches missed to present and perform validations on any existing benchmark models. Merits can be recognized only

when substantial scenarios are computed in real life implementations; and feasible scenarios can be detected comfortably. The techniques that use ant-like agent suffer from redundancy, as ant may traverse the same path repeatedly due to the accumulation of pheromone on some specific routes. The techniques getting inspiration from chromosomal behavior in genetic theory have an inherent randomness that leads to very fewer paths (or scenarios) generated from fork-join.

Figure 2.6 depicts that there exists a variety of techniques under a common umbrella of meta-heuristics. We find that the heuristic in biologically inspired approach named Amoeboid Organism Algorithm (AOA) draws its inspiration from the internal mechanism of the slime mould *Physarum Polycephalum,* which explores the paths from an intermediate entity with lesser redundancy as compared to its peer meta-heuristics like ACO, and GA. The properties in *Physarum* inspired approach are a blend of exploration-based as well as in exploitation-based methodologies [198], [199], [200] that make it suitable among its peers to implement for generating test scenarios from a UML activity diagram.

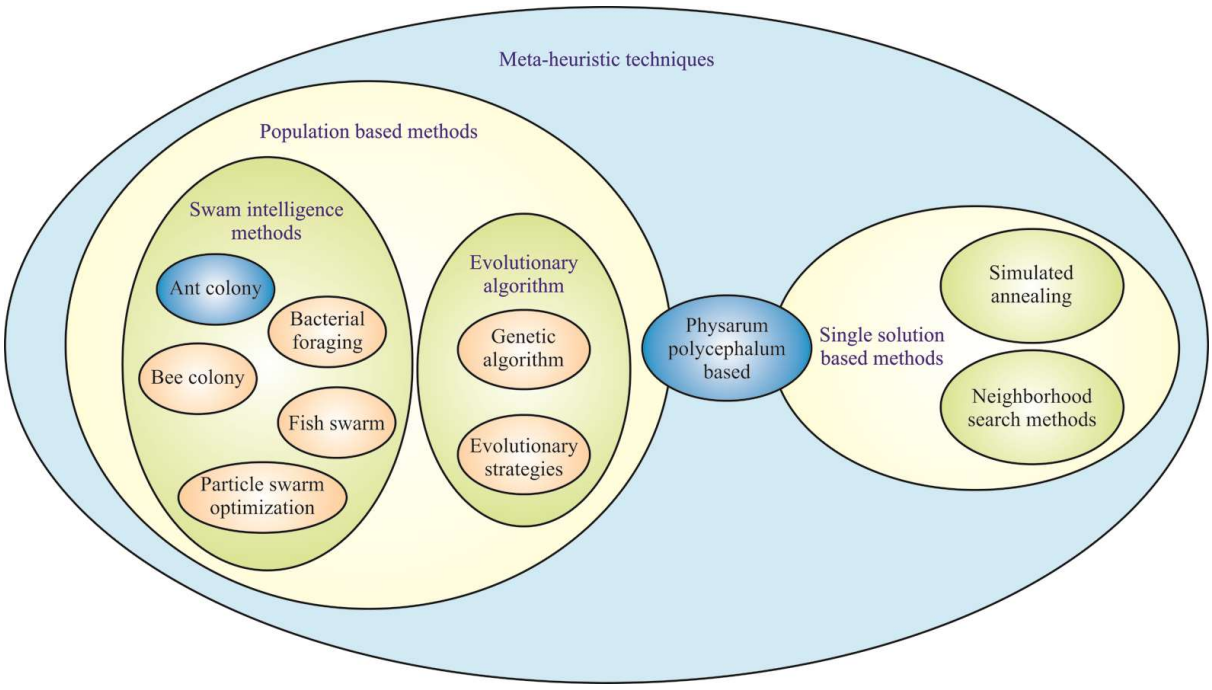


Figure 2.6: Categorization of Meta-heuristic techniques

When compared with the upcoming bio-inspired approaches, the ACO is a well-established meta-heuristic and has already been applied by many researchers in

diversified application areas like supply chain management, route planning, traveling sales man problem, software engineering, software testing [201], knapsack problem, machine design, *etc*. By providing angular movements to the ant like agents in the Ant Colony Optimization (ACO), there can be a better exploration of the search space when compared with its simpler counterpart [202], [203], [204], [205].

The next chapter discusses the proposed amoeboid organism-based algorithm for generating test scenarios from a UML activity diagram.

Chapter 3

Test scenario generation in UML Activity diagram using Amoeboid Organism algorithm

The model-based analysis is receiving a wide acceptance as compared to code-based analysis in the context of prioritizing and guiding the testing effort and speeding up the development process. Ordinarily, system analysts, as well as developers, follow Unified Modeling Language (UML) activity diagrams to render all realizable flows of controls commonly recognized as scenarios of use cases. This chapter is an attempt to apply a bio-inspired algorithm to produce test scenarios for the concurrent section in UML Activity Diagram (AD). Here, the heuristic draws its inspiration from the internal mechanism of the slime mould *Physarum Polycephalum*, a large single-celled amoeboid organism.

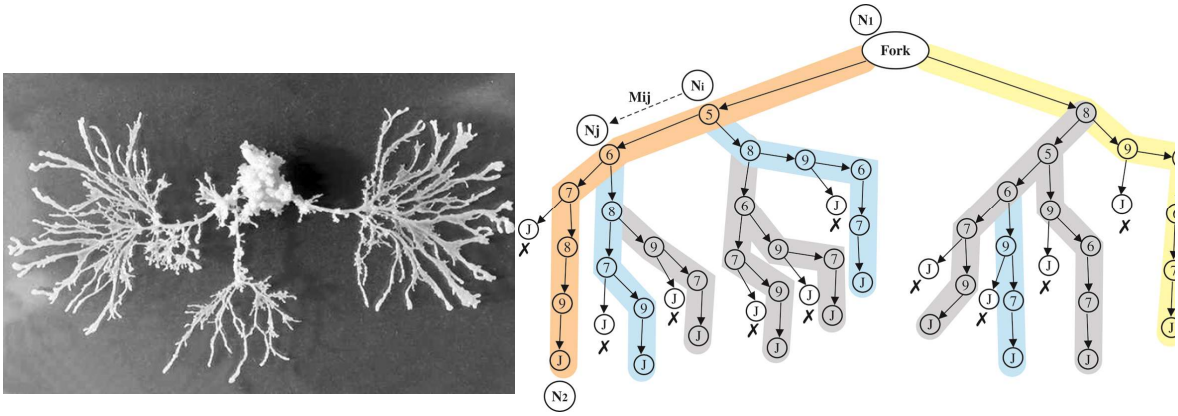
3.1 Amoeboid algorithm and motivation for its usage

In the context of biological taxonomy, the *Physarum Polycephalum* belongs to the super-group Amoebozoa, phylum Mycetozoa, class Myxogastria. It is a single-celled amoeboid organism that shapes a dynamic net of protoplasmic tubes spanning sources of nutrients. The biological system that governs the tube formation and selection leads to the *Physarum*'s path-finding capability: tubes thicken in the direction in which the flux persists. Hence, the plasmodium of *Physarum Polycephalum* can be considered as one of the biological models of natural computation.

The amoeboid organism algorithm is one of the emerging biology-based computational intelligent (CI) algorithms derived from the amoeboid inspired studies [206], [207], [208], [209], [210], [211], [212]. Amoeboid organism exhibits sophisticated computing capabilities where the shape-changing dynamics of such organism includes stochastic fluctuations; and the plasmodium spread by networks can be made programmable. The network of tubes for *Physarum Polycephalum* through which nutrients and chemical signals circulate throughout the organism is shown in Figure 3.1 (a) [213]. Required

circulation for maintaining the appropriate level of nutrients and chemical signals is dependent on streaming across a mixed-up net of Tubular Channels (TCs). Henceforth, the meshed-up and complicated framework of the tubular structure is associated with the carriage of materials and forewarnings within the entity. The tube expands with adequate flux. Summarily, the widening results in the further build up of flux for the opposition to the flow of sol subside in the expanded route. Thus, TCs having a substantial flux flourish, while those having a skimpy flux value vanish.

The amoeboid organism algorithm has already been applied in diversified areas that demand exclusivity of computational abilities such as travel salesman problem, the problem of vehicle routing and solving complex maze [214], [215], [216], [217], shortest path [207], [208], [216], [218], transportation network [209], [219], game theory [220], rough set [221], [222], *etc*. However, to the best of our knowledge, this amoeboid organism approach is yet to be sufficiently researched for imbibing its abilities in solving the problem of test scenario generation for a concurrent section of the UML activity diagram.



(a) (b)

Figure 3.1: (a) Tubular structure of veins in *Physarum Polycephalum* [206], [215]; the (b) Tubular structure in the exploratory tree representing test scenarios among existing permutations.

The pivotal motivator behind the application of amoeboid organism algorithm to look out scenarios for concurrent section in an activity diagram comes from the existing similarity between the tubular veins like structure of *Physarum Polycephalum* and the test scenarios that might be generated by incremental traversal over the edges present in the similar tree type structure representing the permutations under fork-join node. Flux

movements in *Physarum Polycephalum* generate articulated paths from the source node (root node) to sink node (leaf node). Representing a scenario with a single root node and multiple sink nodes consequently developed into a structure, similar to the one shown in Figure 3.1 (a). Marked tubular structure in Figure 3.1 (b) depicts the feasible test scenarios generated by incremental traversal of the exploratory tree structure.

Undoubtedly, the characteristic feature of the duct diameter is auto-synergist. Specifically, the network is adaptable to the fluctuations of flux. The work, therefore, includes this adaptability in the model discussed. Finally, empirical evidence not being extensive, but subjective, there lies a scope of great liberty in devising the model.

3.2 Mathematical model for using amoeboid organism algorithm

The study on the amoeboid organism [223] provides an outline of the tube creation method. TCs thicken in a specified route when shuttle swarming of the protoplasm progresses on that path for a specific time. That means a positive return between flux and the diameter of the tube, as the sol conductivity remains higher in a thicker medium. Using the maze as depicted in [223], a mathematical formulation to generate the test scenarios for a concurrent section in the UML activity diagram has been constructed. Tubular structures are shown in Figure 3.1 (b) represent scenarios under fork-join with two distinct nodes related to the food sources, named as N_1 and N_2, and the other intermediate nodes as N_3, N_4, N_5, so on and so forth. Each time one (N_1) among the source nodes becomes evident as a source; and the other (N_2) imitates a sink. The segment of duct between N_i and N_j signifies an edge as M_{ij}.

3.3 Flux streaming through TCs

The identifier Q_{ij} has been taken for expressing the flux through M_{ij} from N_i to N_j. Assume that the flow over the tubular path is tentatively *Poiseuille flow;* and the same is conveyed by Eq. (1):

$$Q_{ij} = \frac{\pi a_{ij}^4}{8k} \frac{p_i - p_j}{L_{ij}} \quad (1)$$

Where a_{ij} and L_{ij} denote the radius and length of the tube for an edge M_{ij}, p_i specifies the pressure at the node N_i and k is the viscosity coefficient of the sol streaming inside

the tubular structure. By setting $D_{ij} = \frac{\pi a_{ij}^4}{8k}$, as the conductivity of the edge M_{ij}, Eq. (1) can be reframed as under:

$$Q_{ij} = \frac{D_{ij}}{L_{ij}}(p_i - p_j) \quad (1a)$$

Now, taking the law of conservation of sol and assuming zero capacity at each node, it can be summed as:

$$\sum_i Q_{ij} = 0 \quad (j \neq 1, 2) \quad (2)$$

The equations (3a) and (3b) hold for the source N_1 and sink N_2,

$$\sum_i Q_{i1} + I_0 = 0, \quad (3a)$$

$$\sum_i Q_{i2} - I_0 = 0, \quad (3b)$$

Where I_0 is the flux from the source node (or the flux into the sink node). The value of I_0 is considered fixed in the model adopted here which implies a steady total flux throughout the process.

3.4 Adaptation

Experiments conducted in the research works [214], [215], [216], [217] indicate that the tubes with larger flux value continue to exist, whereas the tubes with the reduced flux values deteriorate. Concerning adaptation of tubular width, it is assumed that D_{ij} varies considerably over a time frame as per the flux Q_{ij}. Equation (4) calculates the change of D_{ij}(t), where r is a deterioration rate of the duct.

$$\frac{d}{dt}D_{ij} = f(|Q_{ij}|) - rD_{ij} \quad (4)$$

Equation (4) also means that the conductivity tends to disappear in the absence of flux along the edge, while the conductivity enhances in the presence of flux. Thus, it is likely to presume that f is a monotonically growing continuous function, which equates the value $f(0) = 0$. The edge lengths $L_{ij}'s$ remain fixed all over the adaptation procedure.

Equations (1a)– (3b) have been used to derive the Poisson equation taking pressure as the parameter, as under:

$$\sum_i \frac{D_{ij}}{L_{ij}} (p_i - p_j) = \begin{cases} -I_0 & for\ j = 1, \\ +I_0 & for\ j = 2, \\ 0 & otherwise. \end{cases} \quad (5)$$

The value for all $p_i's$ can be obtained by using Eq. (5); and each $Q_{ij} = \frac{D_{ij}}{L_{ij}} (p_i - p_j)$ is computed by framing $p_2 = 0$ as an elementary pressure level. The present model articulates the evolution process using the variable $D_{ij}'s$ that progresses by the adaptation of Eq. (4). The variables p_i and Q_{ij} are computed by Eq. (5) using the specified value of the $D_{ij}'s$ (and $L_{ij}'s$). The conductivity is proximally attributed to the tubular width. Henceforth, the fading of TCs is stated by the withering of the conductivity of edges. Specific edges flourish or persist, while others droop during the evolution of the model. It is acknowledged that the system has reached at a solution to generate test scenarios when the leftover edges build a path (or paths) bridging the two specific nodes $N_1(source)\ and\ N_2\ (sink)$. Here, the proposed AO-based approach applies flux and conductivity values to decide for the next node to jump upon. Finally, this approach generates a path from source node to sink by incremental traversal over each intermediate edge.

3.5 Proposed approach

In this section, we first list the steps, given in Figure 3.2, for getting test scenarios from UML activity diagram and subsequently, describe our technique. Activity diagram has been drawn using Rational Software Architect (RSA) and exported to its respective Extensible Markup Language Message Interface (XMI) format.

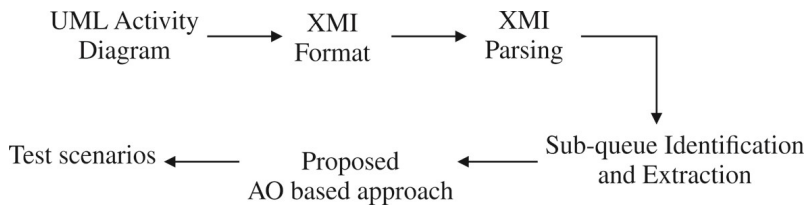


Figure 3.2: Methodology to generate test scenarios from UML activity diagram.

After parsing XMI code, sub-queues present under the fork-join are extracted. The proposed approach using amoeboid organism has been applied to generate the permutations among activity nodes of fork-join sub-queues. To develop the proposed

approach, we have defined some notations which have been used throughout this current chapter.

SQ_i	Sub-queues between *fork-join* structure/s of an activity diagram
a_k	Agent inspired from amoeboid organism
DC_{ij}	Set of nodes directly connected to the node at i^{th} position
$InDPV_k$	Set of child node(s) of the recently visited node(s), where child node(s) belong to the sub-queue(s) different from the one to which i^{th} node belongs
Q_{ij}	Flux between the node i and j
CPC	Current position counter
TN	Number of total activity nodes between fork-join pair
X_k	Set of all unvisited nodes that belong to the sub-queue(s) different from the one to which i^{th} node belongs
F_{iJ}	Set containing nodes that are directly (or indirectly) connected to the current node i
$p_{selection}$	Probability value used to select between directly and indirectly connected nodes
TS_s	Set having test scenarios which start with fork node and end up-to join node.

3.5.1 Algorithm to generate test scenarios for concurrent section

This sub-section outlines the steps for applying amoeboid organism approach to compute the test scenario in the concurrent section of UML activity diagram. The algorithm takes fork-join sub-queues as its input, and an amoeboid organism agent from a particular colony starts traversal from the fork node. A set $DC_{ij} = \{dc_{i1}, dc_{i2} \ldots dc_{ik}\}$ consists of entities directly connected to the fork node. The condition ($i^{th} node \neq fork \; and \neq join$) checks the current node for being an intermediate activity node; and in such a case, it also computes the indirectly connected nodes on which the control may jump. After this step, the probability P_{ij} for all the j^{th} next nodes is computed. A set $InDPV_j$ is computed which constitutes indirectly connected nodes. The condition ($randomNumber \leq p_{selection}$) selects the set of nodes to be chosen by the amoeboid agent, *i.e.*, either directly connected or indirectly connected activity nodes. Under loop construct (**For** $\forall f_{ij} \in F_{iJ}$), the flux Q_{ij} is computed for all the j^{th} elements of the set chosen in the previous step. Flux will be non-zero in two situations, firstly, when j^{th} node(s) is not a join node. Secondly, when it's a join node and all the other available activity nodes in the sub-queues have already been covered, and flux is zero, otherwise. A set Q_{iJ} constitutes the flux value for all the j^{th} elements. When the computed flux value comes out to be zero, the edge from the current i^{th} node to next j^{th} node is

rejected. If the flux is non-zero, and the j^{th} node is join, then accept the edge from i^{th} node to j^{th} node. The whole path from fork up-to-current join node is accepted as a feasible path. When flux is non-zero, and j^{th} node(s) is not a join node, then replicate (or clone) the amoeboid agent on the j^{th} node(s) and these j nodes will become current node as i. From all i node(s) repeat the initial step to compute the directly connected nodes and execute till the stopping criteria are met. By taking into consideration the steps as mentioned above, the algorithm results into a sub-set (slice) of feasible paths, and neglects the infeasible ones.

Input: Sub-queues(SQ_i), where $i = \{1,2,3 \dots n\}$, TN
Output: Test scenario(s)(TS_s): $ts_1, ts_2, \dots ts_m$

Begin

1. Initiate the traversal of a_k from *fork* node at i^{th} position with CPC_i
2. Compute $DC_{ij} = \{dc_{i1}, dc_{i2} \dots dc_{ik}\}$
3. **If** $i^{th} node \neq fork \; and \; \neq join$ **then**

$$\forall\, j \in X_k,\; compute\; P_{ij} = \begin{cases} \frac{Q_{ij}}{\sum_{j \in X_k} Q_{ij}} & if\; j \in InDPV_k \\ 0 & if\; j \notin InDPV_k \end{cases} ,Where\; InDPV_k \subseteq X_k$$

$$InDPV_j = j^{th}\; next\; node(s)\; with\; non-zero\; probability\; value$$

Else

$$InDPV_j = \emptyset$$

EndIf

4. **If** $randomNumber \leq p_{selection}$ **then**

$$F_{iJ} \leftarrow DC_{ij}$$

Else

$$F_{iJ} \leftarrow InDPV_{ij}$$

EndIf

5. **For** $\forall\, f_{ij} \in F_{iJ}$ **do**

$$CPC_{j^{th}\, node} = CPC_{i^{th}node} + 1$$

$$\Delta p_{ij} \begin{cases} = 1,\; If\; j^{th} node \notin join \\ = 1,\; If\; ((j^{th} node \in join) \wedge (CPC_j = TN_{fj} + 1)) \\ = 0,\; If\; ((j^{th} node \in join) \wedge (CPC_j \neq TN_{fj} + 1)) \end{cases}$$

$$Q_{ij} = \frac{\pi a_{ij}^4 \Delta p_{ij}}{8k\, L_{ij}}, \text{ where } Q_{ij_m} \in Q_{iJ} = \{Q_{ij_1}, Q_{ij_2}, \dots \dots \dots \dots Q_{ij_n}\}$$

EndFor

6. **For** $\forall\; Q_{ij} \in Q_{iJ}$ **do**

If $\exists\, Q_{ij_m} = 0$ **then**

$$neglect\; edge\; i \rightarrow j$$

$$\therefore neglect\; full\; path\; from\; fork\; up\; to\; join\; node$$

Continue

EndIf

If $\exists\ (Q_{ij_m} \neq 0\ \wedge j\ \notin join)$ **then**
$accept\ all\ edges\ from\ i \rightarrow j$
$replicate\ the\ agent\ a_k\ on\ j^{th} node(s)$
$node\ j\ will\ become\ the\ current\ node\ now\ (i \leftarrow j)$
$For\ all\ j^{th} node(s);\ Move\ to\ step\ 2$
Continue
EndIf

If $\exists\ (Q_{ij_m} \neq 0\ \wedge\ j\ \in join)$ **then**
$accept\ edge\ i \rightarrow j$
$Path\ from\ fork\ to\ current\ join\ node\ will\ result\ into\ feasible\ \boldsymbol{ts_z}$
$Load\ ts_z\ in\ set\ TS_s$
Continue
EndIf
EndFor

3.5.2 A sample run of the proposed algorithm

Figure 3.3 represents the sample run of the proposed approach on the concurrent section of activity diagram in Figure 2.3 of Chapter 2. In Figure 3.3, the solid line denotes directly connected; and dashed line represents the indirectly connected node. Cloud like symbol denotes an amoeboid organism agent. While simulating the proposed approach the initial value for the parameter CPC_i and $p_{selection}$ is taken as 0 and 0.8 respectively.

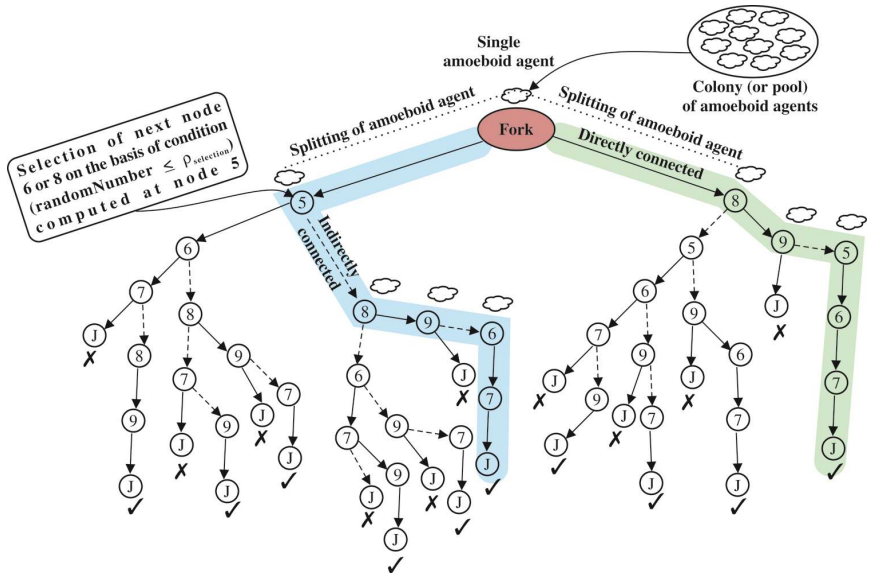


Figure 3.3: Sample run of the proposed approach on an example activity diagram.

The algorithm commences with the traversal of a single agent a_k, from a particular colony, initiated at the fork node. From the current position, a set DC_{ij} is computed, which contains the nodes that are directly connected to the current node. This set may

contain more than one element. Using *Poiseuille flow* (Eq. 1), elements for set $InDPV_j$ are computed. $p_{selection}$ stands for a fixed number probability value which is used to compare with a randomly generated number. This comparison selects the tentative element(s) for agent a_k to jump upon by choosing among directly and indirectly connected elements. Flux is computed for the elements of chosen set. When the computed flux for the element(s) is non-zero, and the same is not a join node, then edge between $i \rightarrow j$ will be accepted. Agent a_k will get virtually replicated on each j^{th} element; and flow is repeated with the computation of set DC_{ij}. A particular thread of an agent stops traversing when a join node is encountered to an associated non-zero flux value. Using a single agent here, two paths are obtained, namely, $Fork \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 6 \rightarrow 7 \rightarrow Join$ and $Fork \rightarrow 8 \rightarrow 9 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow Join$.

3.6 Experimental results and analysis

3.6.1 Parameter setting and stopping criteria

To simulate the work, the considered experiments have been performed on Intel Core i3-6100U, 2.30GHz, 4GB RAM, Windows 10, 64-bit operating system, which has been referred as configuration C1. Guidelines provided in the study by Črepinšek *et al.* [224] have been referred for setting the values of control parameters while simulating existing ACO, GA, and the proposed AOA approaches. Further, the parameters have been set to different algorithms as follows:

- GA parameters

GA has already been applied in multiple sub-domains such as gene theory, genetics, multiple sequence alignment, bio-informatics, knapsack problem, software optimization, *etc.* for sequence generation and/or optimization [225], [226], [227]. Here, AOA has been compared with EA (that is a version of GA) of Shirole *et al.* [189]. Figure 3.4 outlines the approach adopted for simulating EA along with control parameters.

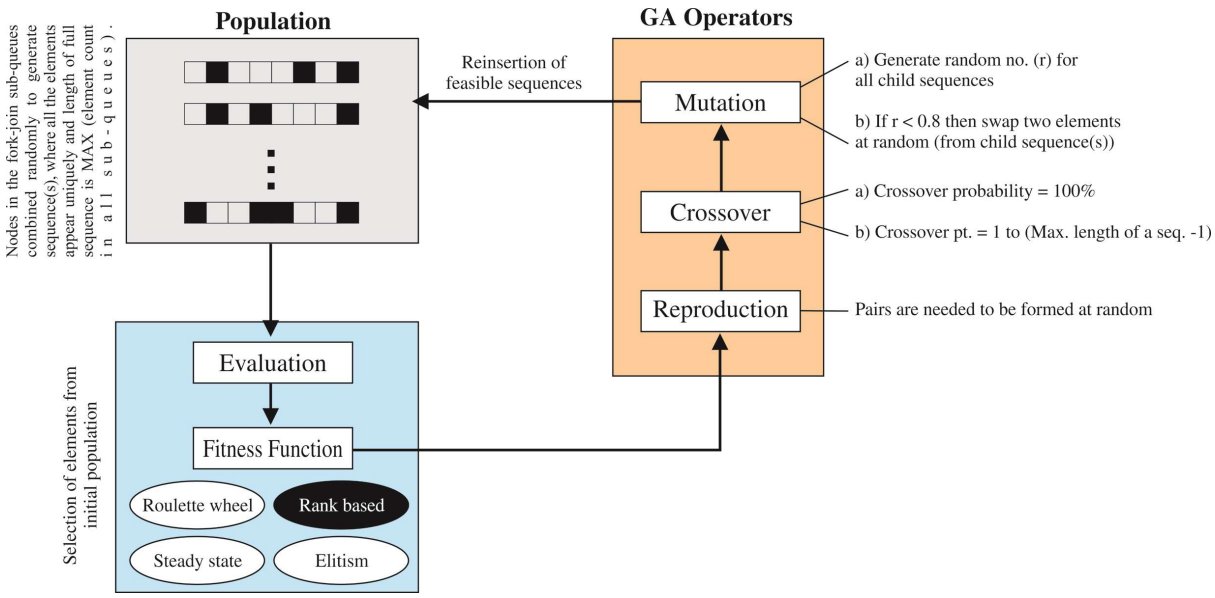


Figure 3.4: Generating test scenarios through EA approach [216], [223].

- ACO parameters

The technique of using ant colony optimization for generating paths was provided by Srivastava *et al.* [192]; and the same has been simulated to have a comparison with the proposed approach to this work. Existing ACO-based method [192] uses the fundamentals as derived from the study conducted by Dorigo *et al.* [228]. Various parameters with their associated values taken for simulating ACO are as follows: τ_{ij} is the pheromone level at the edge from node $i\ to\ j$ (0.8), ρ being the evaporation rate of pheromone (0.1), η_{ij} heuristic value for the edge between node $i\ to\ j$ (1.5), α factor controlling the pheromone level (0.9), and β factor controlling the heuristic value (0.8).

- AOA parameters

Various parameters with their associated values taken for simulating the technique using amoeboid organism for generating the test scenarios are as follows: CPC_i for an agent a_k at *fork* node is 0; and value for $p_{selection}$ is taken as 0.8.

- Stopping Criteria

Various bio-inspired approaches (e.g. ant colony, genetic algorithm) usually engage varied amount of fitness evaluation functions. Researchers in their studies [229], [230] suggested considering an equal number of consumed fitness function evaluations to

achieve a fair comparison amongst different meta-heuristics. Following this recommendation, we set a maximum number of fitness evaluation function equal to 300000 as stopping criteria for all the algorithms (AOA, GA, and ACO) simulated here for performing the experimentation.

- Generic settings for all the algorithms

For conducting an empirical analysis, the computation efforts for the meta-heuristic can be used to compare the given instance [185]. To execute a defined set of statements for the given iterations, computational effort is the total execution time taken by the machine. The computational effort of the proposed AOA has been analyzed by considering two factors, namely, (i) Time in milliseconds for the execution of algorithm only, and (ii) Time in milliseconds to convert UML activity diagram into XMI and further up to the execution of the algorithm. In the experimental set-up, where AOA, GA, and ACO are simulated, the parameter named the number of agents in AOA is considered as equivalent to the population in GA or number of ants in ACO. The number of trials in AOA is considered same as the number of generations in GA or number of trials in ACO. All the approaches are executed thirty times for each subject system. For executing the algorithms on C1, the trial count is taken as 10 and population as 100. For comparing the computational effort, the time (in milliseconds) to execute the algorithm only has been considered as the parameter.

3.6.2 Objectives of experimentation

The proposed approach is executed using the activity diagrams taken from the LINDHOLMEN dataset [231], student projects, and few synthetic cases. The prime intents of experimentation are as follows:

(i) To compare the proposed approach with ant colony and genetic algorithm by the count of feasible test scenarios (or sequences) generated.

(ii) To conduct statistical analysis for validating the results obtained through the amoeboid organism approach.

3.6.3 Subject systems

Since the proposed approach aims to produce feasible test scenarios in design level UML activity diagrams, accessibility of UML design level requirements is quite essential for

its validation. In the product-based software industry, design level activity diagrams having sufficient details are prepared before the coding phase for software (or system). However, such types of designs are not obtainable in the open world for the sake of privacy and sensitivity concerns, and that is why these could not be exercised in the present research. Hence, the LINDHOLMEN dataset [231] was used which provided a list of open source projects that used UML. A flowchart presenting modus operandi for selecting activity diagrams has been taken from [231] and is shown in Figure 3.5. The details of finally selected activity diagrams are given in Table 3.1.

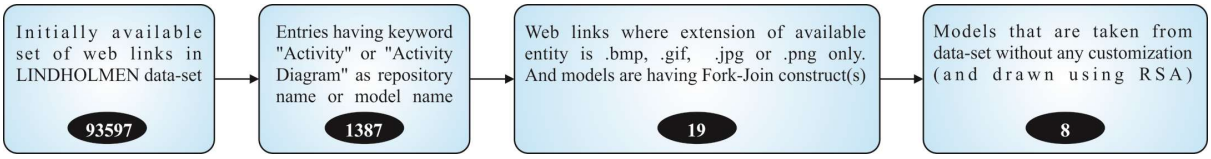


Figure 3.5: Selection procedure of activity diagram from the LINDHOLMEN data-set.

Table 3.1: Activity diagrams from the LINDHOLMEN data-set selected for experiments.

Model No.	**Repository name in LINDHOLMEN data-set**	**# of elements in AD**	**# of fork-join(s) in AD**	**# of elements under fork-join(s)**	**Control constructs present in fork-join**	**# of scenarios generated using** [189]
LDSET1	267492/moje_rep/	34	1	18	Y	120
LDSET2	atadeesom/BubbleBeeDoc/	31	1	10	Y	6
LDSET3	BGCX261/zimeoprocess-svn-to-git/	65	2	6 (FJ1), **20(FJ2)**	Y	20
LDSET4	DmitriyG/Diplom/	36	1	15	N	120
LDSET5	HandreWatkins/COS-301-Phase2/	38	1	24	Y	1680
LDSET6	ooad-2014-2015/JotaPlanet/	33	2	**9(FJ1)**, 6(FJ2)	Y	7
LDSET7	85pando/Zusammenfassungen/	22	1	16	N	35
LDSET8	bsaunder/cs414-discussions/	56	3(1, 2*)	**11(FJ1)**, **34**(FJ2,contain nested FJ3 with 6 elements)	Y	9, 13

Model numbers LDSET1, LDSET2, LDSET4, LDSET5, and LDSET7 have one fork-join (FJ) node. LDSET3 and LDSET6 contain two fork-join pairs, but we have considered the fork-join having more number of elements in it. Model number LDSET8 constitutes three fork-join constructs, one independent (FJ1) and one with nested fork-join (FJ2, FJ3; where FJ3 is the part of FJ2). The current work refers to independent

fork-join (FJ1) of LDSET8 as LDSET8 (a), and another (FJ2) as LDSET8 (b) while using these in the experiments.

3.6.4 Comparison of AOA with existing GA and ACO

In this sub-section, benchmarks, as identified from the LINDHOLMEN dataset, are taken for comparing the proposed AOA with the existing GA and ACO.

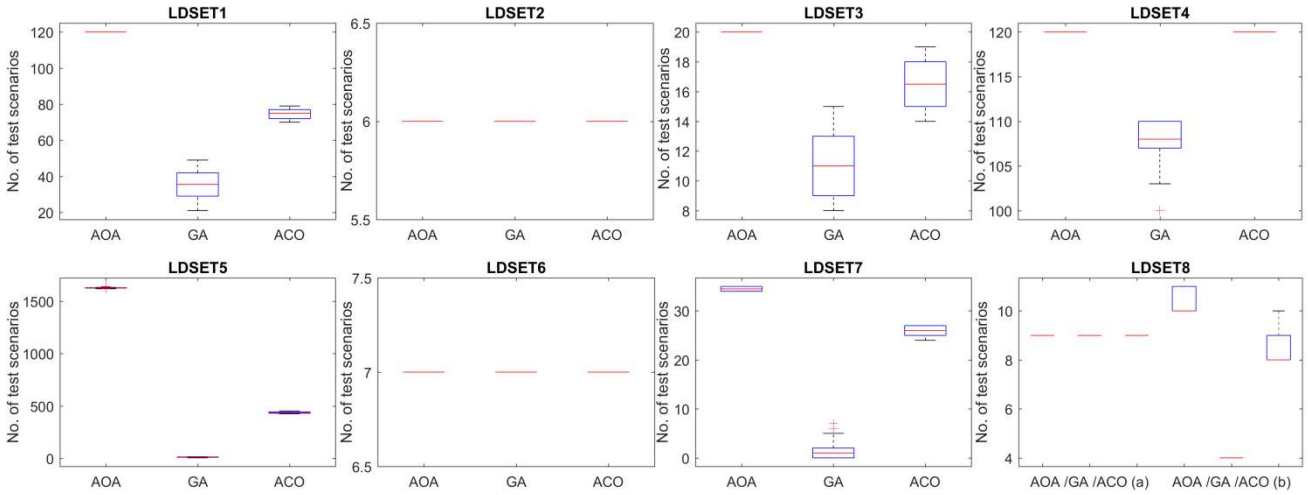


Figure 3.6: Generation of feasible test scenarios using the AOA, GA and ACO.

For the cases taken from the LINDHOLMEN dataset, the box-plot shown in Figure 3.6 depicts that the AOA is providing more number of feasible test scenarios as compared to GA and ACO for LDSET1, LDSET3, LDSET5, LDSET7, and LDSET8b. For the benchmarks LDSET2, LDSET6 and LDSET8a, AOA is providing results similar to those of GA and ACO. For LDSET4, AOA is providing results similar to those of ACO, but better as compared to GA results.

3.6.5 Student project as subject system

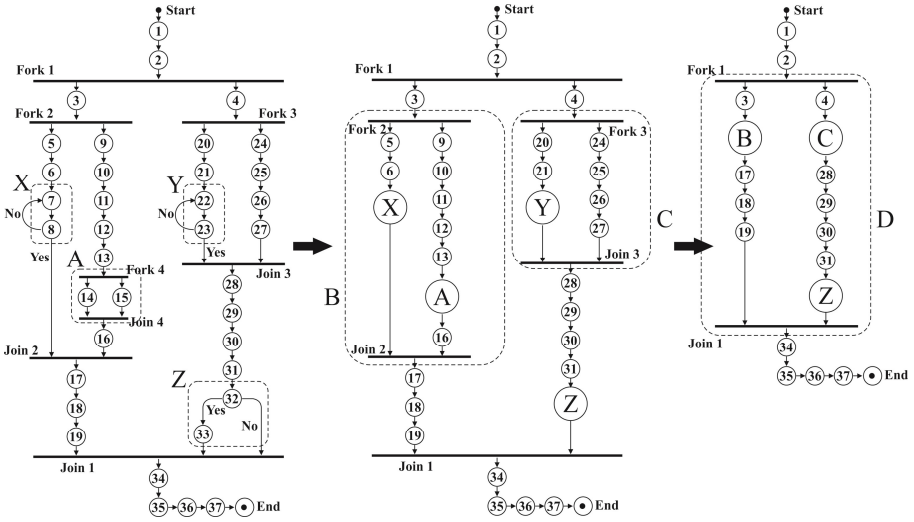
In addition to the LINDHOLMEN dataset, liberty has been availed to use the models created by eight bachelor of engineering (B.E.) students as a part of their academic assignment in the software engineering class tutored by the penman of this thesis in the year 2015, and the same have been further fine-tuned [232]. The students worked together in two concurrent groups of three and five participants respectively, each spending around 45 working hours on devising and creating the models with its corresponding contextual description. All the students followed the same methodology as

discussed during the course. The method used was at par with the one being employed in the industrial projects. The students used RSA with a significant portion of UML 2.2. Table 3.2 describes the size measures for the two activity diagram models created by the student groups. Here, the count of elements under fork-join is tentatively three times as compared to almost all the cases of LINDHOLMEN dataset. Thus, the models under student project also served a great purpose to test the scalability of the proposed approach.

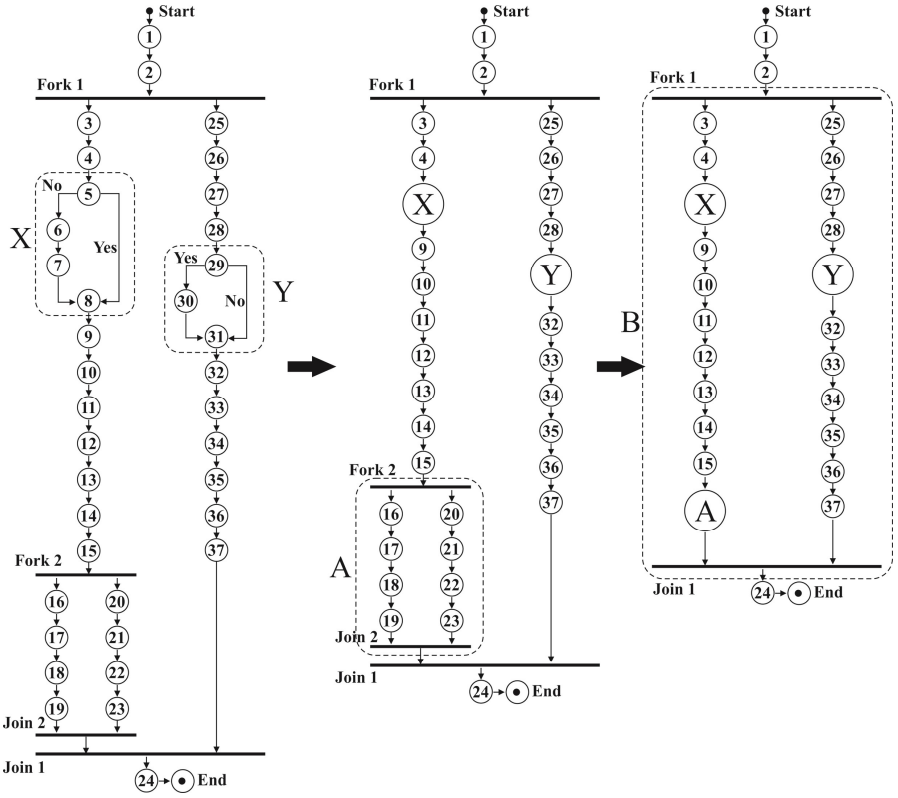
Table 3.2: Size measures of the activity diagrams with number of total elements.

Model No.	AD Model [232]	# of students involved	# of elements in AD	XMI File (KB)	# of fork-join(s) in AD	# of control construct(s) in AD	Nested Fork-Join
STDP1	**Manchurian**	3	98	29	4	3	Yes
STDP2	**VideoFile**	5	89	26	2	2	Yes

All the activity diagrams with nested fork-join structure and control constructs like the loop, if-else, *etc.* have been transformed first into an Intermediate Testable Model (ITM) that serve as an input activity diagram with no nested/control construct [190]. To get the desired simplified activity diagram, each one of the nested fork-join structure or the control construct has been replaced by a single annotated node. Starting from the innermost fork-join (or control construct) the replacement heads towards outer construct until there remains only one fork-join. Here, the proposed AOA has been applied to each independent and simplified activity diagram having concurrency construct (with no nested or control constructs) for generating the test scenarios. Figure 3.7 highlights the whole transformational process of receiving a simplified activity diagram from the complex nested version of activity diagrams which have been taken from [232]. Consequently, the resulted final activity diagram remains 'the derived final non-nested simplified version' generated to serve the experiment.



(a)



(b)

Figure 3.7 (a) and (b): Transformational process of the activity diagram titled Manchurian and VideoFile respectively.

Table 3.3 shows the count of scenarios that can be generated using the formula in [189] for each sub-section of the graph shown in Figure 3.7 for STDP1 and STDP2.

Table 3.3: Count of test scenarios generated through the formula described in [189].

Model No.	Model Name	Annotation for the sub-section in Figure 3.7	# of scenarios generated using [189]
STDP1	Manchurian	A	2
		B	120
		C	35
		D	792
STDP2	VideoFile	A	70
		B	705432

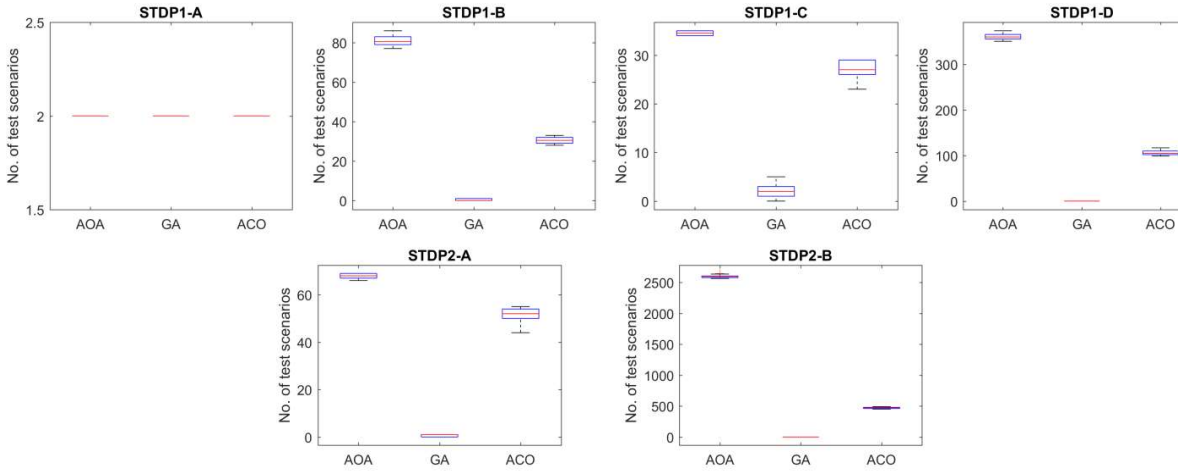


Figure 3.8: Count of feasible test scenarios generated using AOA, GA and ACO on student project 1 (STDP1) and student project 2 (STDP2).

The box-plot shown in Figure 3.8 explains that the AOA is providing more number of feasible test scenarios as compared to its meta-heuristic peers, GA and ACO for the sub-activity diagrams (A, B, C, D for student project 1; and A, B for student project 2) of Figure 3.7. The application of AOA on the final simplified version of activity diagram has resulted into generating feasible test scenarios with a lot of annotated nodes (each representing nested fork-join or control construct). AOA is recursively applied to generate further feasible test scenarios on annotated nodes representing nested fork-join. Annotated nodes representing control constructs get substituted by considering its all simplified paths from start to the end node [190]. By doing the substitution for annotated nodes, the final count of feasible test scenarios increases greatly.

In the student projects considered here, GA is performing ghastly because of inherent randomness at crossover and mutation phase in the said algorithm. The experiment shows that when node count in the sub-queues under fork-join increases, the randomness

factor in the algorithm will lead to an erratic combination of the nodes in the final scenarios.

3.6.6 Comparison of proposed AOA with DFS on large search space

Table 2.7 shows that the use of DFS is not worth for the activity diagrams where the count of test scenarios is large. To justify this statement, DFS was compared with the proposed AOA by prematurely ending the exact algorithm for sub-activity diagram B of STDP2, where the count of test scenarios generated using the formula [189] is huge. The time taken for premature termination of DFS is the same as by the proposed AOA (using the same parameter settings as taken earlier while comparing AOA with its peers, except the maximum count of fitness function evaluations which had been raised to 3000000). Figure 3.9 reflects that AOA is better as compared to DFS for the UML activity diagrams where the count of generated test scenarios is quite large.

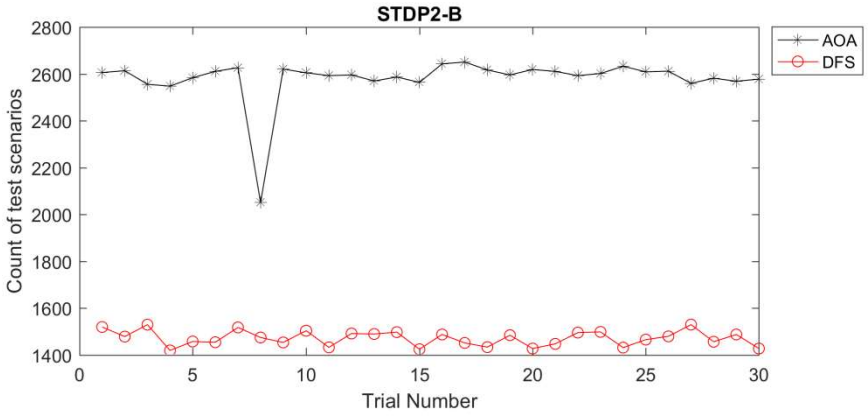


Figure 3.9: Count of test scenarios generated through AOA and DFS using STDP2-B on C1.

3.6.7 Activity diagram with multiple join nodes

In this sub-section, Figure 3.10 exhibits the synthetic cases for activity diagrams having one fork node and multiple join nodes.

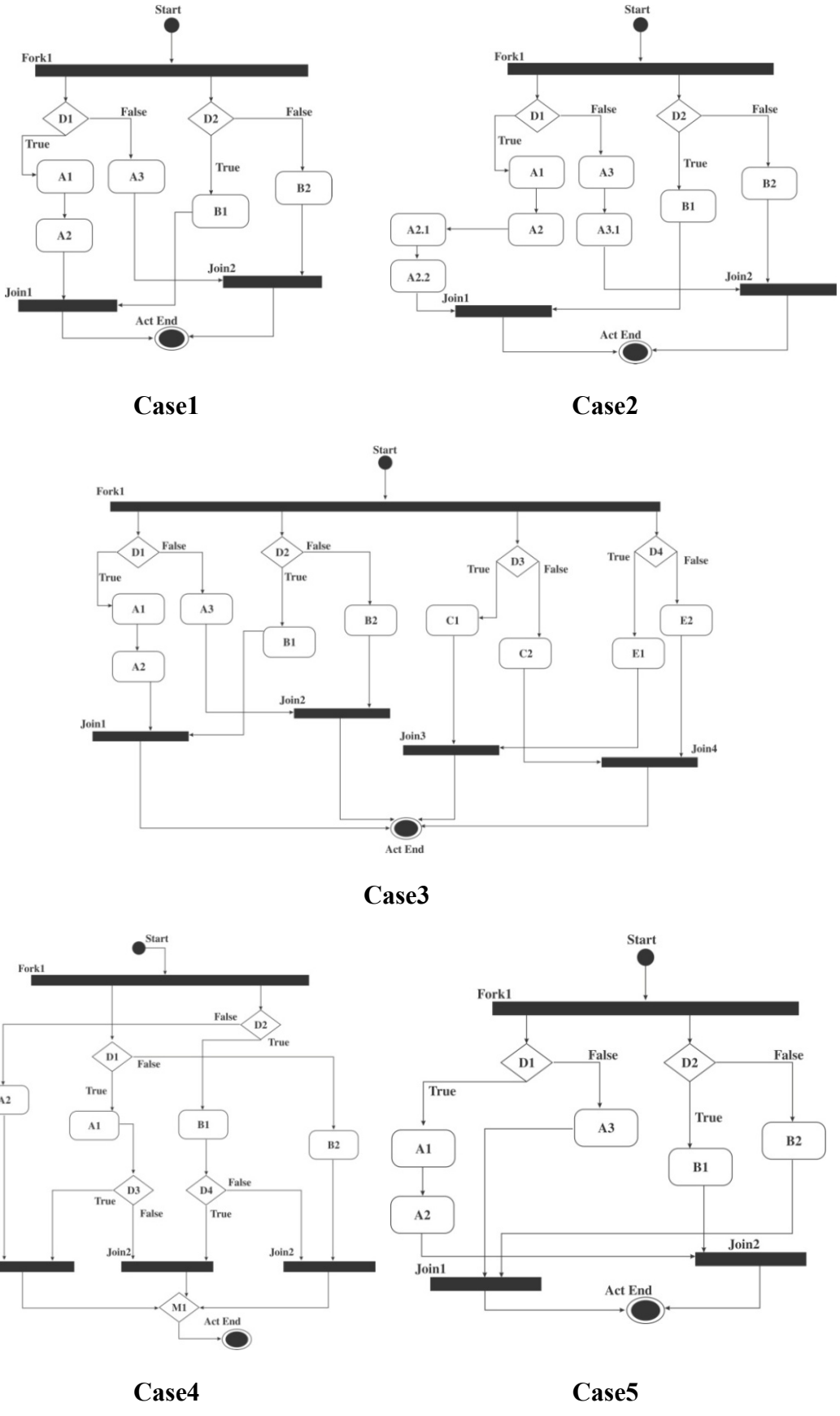


Figure 3.10: Synthetic activity diagrams with single fork and multiple join nodes.

Line graph as shown in Figure 3.11, and box-plot as presented in Figure 3.12 highlight the results obtained after executing AOA, GA, and ACO on synthetic activity diagrams (as shown in Figure 3.10).

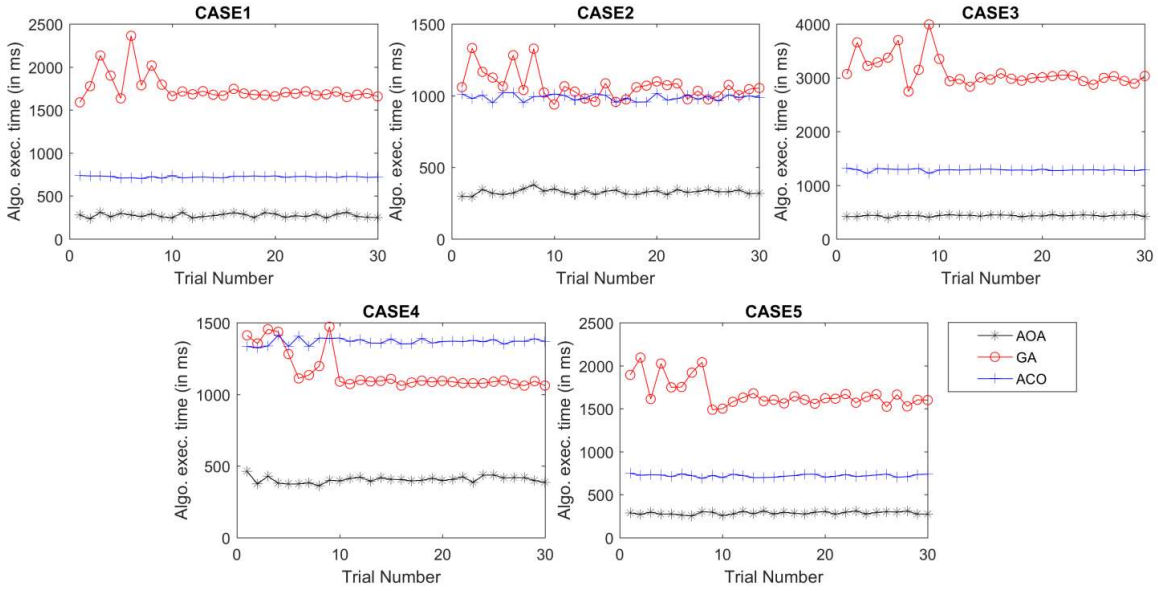


Figure 3.11: Computational effort (in millisecond) for AOA, GA and ACO (algorithm execution only).

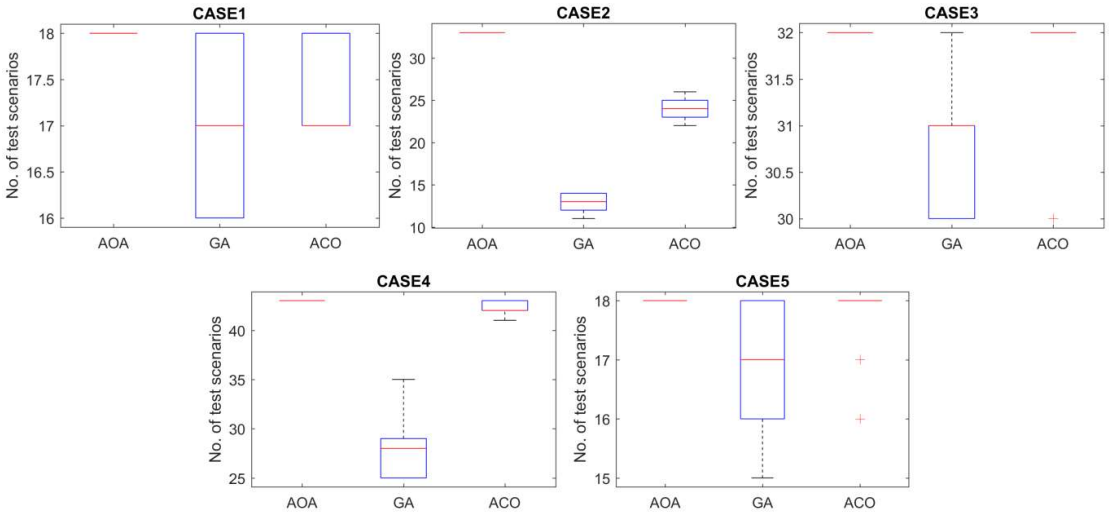


Figure 3.12: Generation of feasible test scenarios after the application of AOA, GA and ACO.

The results highlighted in Figures 3.11 and 3.12 indicate that AOA is either better or equivalent to GA and ACO about the number of feasible test scenarios obtained. Further, AOA is found to be better than GA and ACO concerning the time consumed in milliseconds for executing the algorithm.

3.7 Statistical Analysis

Keeping in view the research studies conducted by Derrac *et al.* [233] and Garg [234], t-test was performed for the purpose of this study using MATLAB tools on the pair of algorithms (using the count of test scenarios obtained) to examine the statistical significance of the results as highlighted in Figure 3.6. Further, statistical evaluation has been done with the presumption that the populations have identical variances at the significance level of $\alpha = 0.05$ for the presented AOA outcomes with ACO and GA results.

Tables 3.4 and 3.5 present the t-test results considering parameters h, p, SD and Mean; where, h is hypothesis test result; p is the probability of observing a test statistic as extreme as or more extreme than the observed value under the null hypothesis; SD is the standard deviation; and mean is the average of results obtained for all the instance runs. Entry, Not a Number (NaN) for h and p in the t-test, denotes that the values taken by comparing the results obtained from the pair of algorithms are the same for all the instances. Thus, SD comes out to be zero in this case.

Table 3.4: Results of t-test taking count of feasible test scenarios generated as a parameter for AOA and GA.

Model No.		**AOA Vs GA**					
		h	**p**	**SD AOA**	**SDEV GA**	**Mean AOA**	**Mean GA**
LDSET1		1	1.92E-57	0	6.683398544	**120**	35.56667
LDSET2		NaN	NaN	0	0	6	6
LDSET3		1	1.72E-30	0	2.063364065	**20**	11.46667
LDSET4		1	1.20E-36	0	2.157318472	**120**	108.3667
LDSET5		1	8.78E-135	5.144353003	2.896886954	**1628.133333**	11.23333
LDSET6		NaN	NaN	0	0	7	7
LDSET7		1	5.94E-66	0.508547628	1.792705526	**34.5**	1.4
LDSET8	a	NaN	NaN	0	0	9	9
	b	1	1.49E-58	0.479463301	0	**10.33333333**	4

Table 3.5: Results of t-test taking count of feasible test scenarios generated as a parameter for AOA and ACO.

Model No.	**AOA Vs ACO**					
	h	**p**	**SD AOA**	**SDEV ACO**	**Mean AOA**	**Mean ACO**

Model No.		AOA Vs ACO					
		h	**p**	**SD AOA**	**SDEV ACO**	**Mean AOA**	**Mean ACO**
LDSET1		1	1.68E-65	0	2.62722265	**120**	74.16667
LDSET2		NaN	NaN	0	0	6	6
LDSET3		1	1.05E-17	0	1.52224879	**20**	16.6
LDSET4		NaN	NaN	0	0	120	120
LDSET5		1	1.14E-116	5.144353003	7.308002227	**1628.133333**	435.8
LDSET6		NaN	NaN	0	0	7	7
LDSET7		1	2.76E-46	0.508547628	0.949894126	**34.5**	25.83333
LDSET8	a	NaN	NaN	0	0	9	9
	b	1	3.74E-12	0.479463301	0.858359837	**10.33333333**	8.766667

The test has been carried out antagonistic towards the null hypothesis that there exists no distinction in their population means. Other than NaN, the value of h as **one** denotes rejection of the null hypothesis. In each case, the p-values obtained are less than the significance level $\alpha = 0.05$. Hence, there exists a difference between the two types of means. Apart from it, the average of the test scenarios obtained for the proposed AOA approach is better, and Standard Deviation (SD) for the results obtained using AOA is less than the existing ACO and GA. Thus, the results are statistically significant.

3.8 Threats to Validity

The prospective threats posed to the validity of subject system used have been discussed here. There are three types of such threats as described below:

- The threats to **construct validity** deal with the issue of measurement and raise concerns about the things to be measured and ignored in simulations. Failing the same will pose serious threats to validity. The primary threat to construct validity is where test sequence generation techniques using AOA involve flux (Q_{ij}) value calculations. Flux values play a significant role in deciding the next node to be added in the sequence; and thus, it is a vital deciding factor for full sequence generation. We use some static initial values for the Q_{ij} constructs, whereas some other estimates may turn to be of greater significance to enhance the efficiency of AOA.

- The threats to **internal validity** are concerned about the changes, which if introduced will affect the outcome. A major concern with regard to internal

validity in such studies has been instrumentation effects, failing which they may lead to the biased results. To minimize and control the resemblance of similar effects, the validation is performed by algorithmic execution on benchmark and synthetic activity diagrams. One source of such effects is the difference in the test process inputs, in which the locality of the model varies. At this time, the effects related to the locality of model changes (*i.e.* cross-synchronization, automatic detection of missing nodes, *etc.*) are not considered. To limit the problems associated with this, the proposed algorithm was applied to each subject model to validate the results for scalability.

- The threats to **external validity** of the present study are based on the issues of how pivotal the subject systems of our studies are. The subject models taken from LINDHOLMEN dataset and student projects are of either small or medium in size. The models, larger in size, may yield flexible cost-benefit trade-offs. The decision to use a single type of enhancement at a time in the model to check the scalability of the algorithm may involve another source of threats. Although the single type of enhancement facilitated the process of simulation and analysis, yet in practice, the model extension might crop up in various other distributions. In general, such threats can be addressed only by conducting further studies on additional subjects.

3.9 Conclusion

After having an analysis of the experimental evaluation of AOA on UML activity diagram, it has been observed that important concern crops up in case the sub-queue(s) under fork-join interleaved with the execution of the respective adjacent sub-queue(s). The same creates a sequence explosion. At present, the studies of amoeboid organism-based test sequence generation techniques are pre-eminent to researchers and advanced research on heuristic approaches is pivotal to determine resource-feasibility. Studies can be empowered and enriched by opting amoeboid organism-based approaches. The AOA is an excellent alternative as compared to GA and ACO to find the feasible set of scenarios when elements in the set of total generated paths are affluent.

The next chapter discusses the proposed orientation-based ant colony algorithm for generating test scenarios for concurrent section of a UML activity diagram.

Chapter 4

Test scenario synthesis using Orientation-based Ant Colony in UML Activity diagram

In model-based development paradigm, the design and testing activity can be initiated in the early phase of SDLC that results in resource reduction regarding time and labor. Further, traceability maintenance among test scenarios and design models is easy, which results in an easy adoption of the speedy modifications in technology and operational environment. In the design phase, the UML activity diagrams are used as a *de facto* standard for workflow representations, sequence ordering, and concurrency. The present chapter shows the implementation of a nature-inspired algorithm used for generating test scenarios in UML activity diagram for the fork-join entity. Here, the heuristic takes its motivation from the pheromone-based path traversal by the ants from their nest to food source.

The naturalists documented the behavior of ants for the first time in the late 1960s and early 1970s. Ant Colony Optimization is one among the most accepted meta-heuristic techniques. It is inspired from natural Ant system (AS), which is prototyped on the food searching behavior of real ants [235]. Ants are treated as the primitive agents in AS, which construct the solutions using heuristics. The reason for choosing ACO among its peer meta-heuristic algorithms like Tabu Search, Simulated Annealing, Genetic Algorithm, Firefly Algorithm, Bee colony, Artificial Bee Colony, *etc*. is two-fold [236]:

- Utility of most of the existing meta-heuristic approaches is without having any convergence proof and results are based on experimentation only. Whereas, analytical proof for convergence in ACO was provided by Gutjahr [237] in the year 2000.

- For solving an optimization problem, the analogy of ants could be easily understood and related to the system under consideration.

4.1 Ant colony algorithm and motivation for its usage

Path taken by the ants from the nest to food source is always the shortest among the set of available paths. Researchers found that ants follow the trace of pheromone laid by the ants that have taken the same path earlier. Further research by Deneubourg *et al.* [238] on AS supported that the probability for following a path is dependent on pheromone as well as the heuristic factor. Dorigo *et al.* [239] proposed usage of the ant analogy related to the behavior of searching for the food in optimization algorithms; and it's a fusion of greed heuristic, positive feedback, and distributed computation.

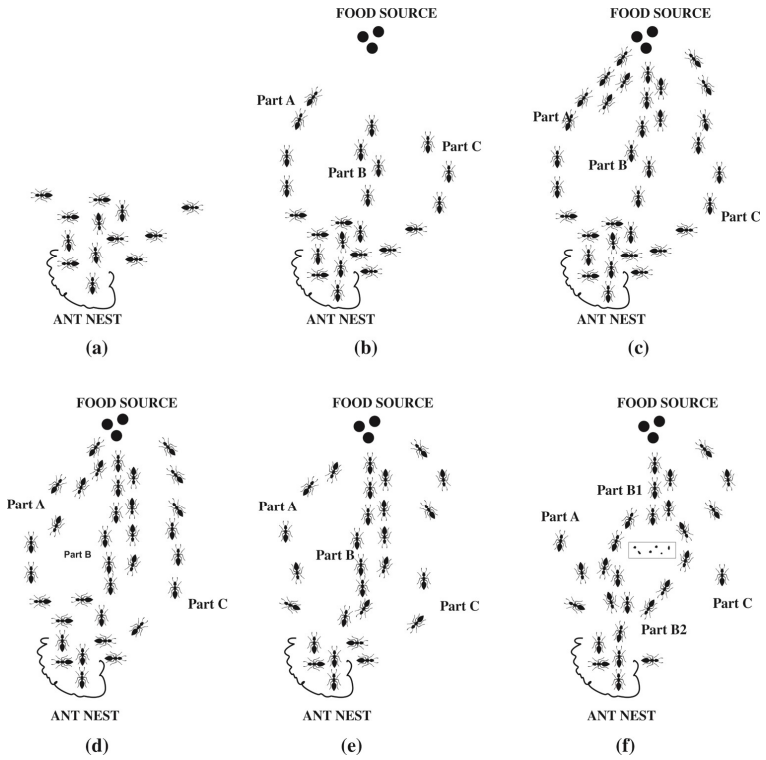


Figure 4.1: Schematic of foraging behavior in ACO [240].

Figure 4.1 (a) and (b) show the random selection and traversal of the possible route from the initial point (ant nest) to the food source (destination). Figure 4.1 (c) depicts the backward route from food source to initial nest position. Ant form "pheromone trails" by depositing a chemical substance called "pheromone". Trailing of pheromone by other ants in the colony has been shown in Figure 4.1 (d). As shown in Figure 4.1 (e) successive ants smell the deposited chemicals and traverse the route marked by rich pheromone concentration. In Figure 4.1 (f), traversal of the route by the ants is self-adjustable and correctable, even in the presence of unforeseen obstacles. The collective

behavior of ants increases the tendency of path selection with the count of ants having selected a specific path in the previous steps. The AS methodology uses intelligence gained by one artificial agent during its search process for constructing the next feasible (or optimal) solution in the domain of the path-traversing problem.

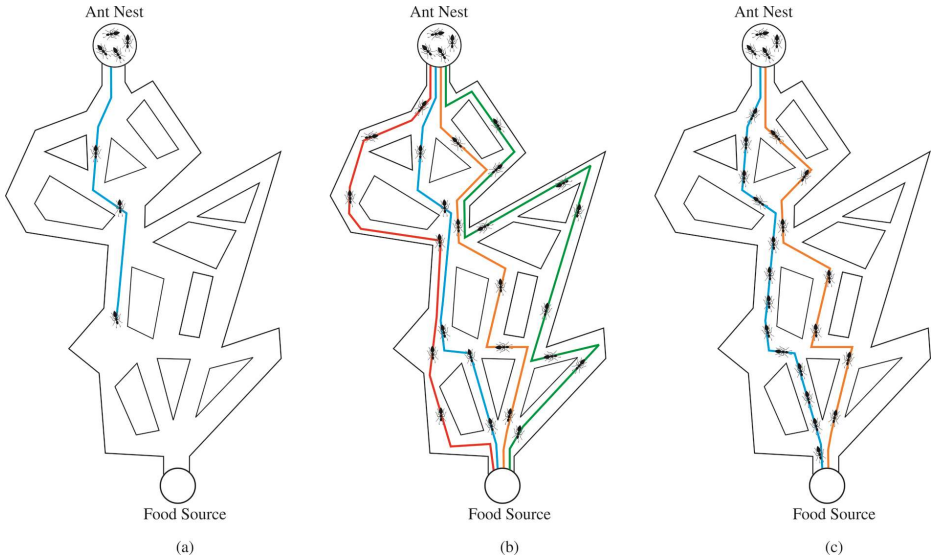


Figure 4.2: Path traversal by Ants from nest to food source [240], [241].

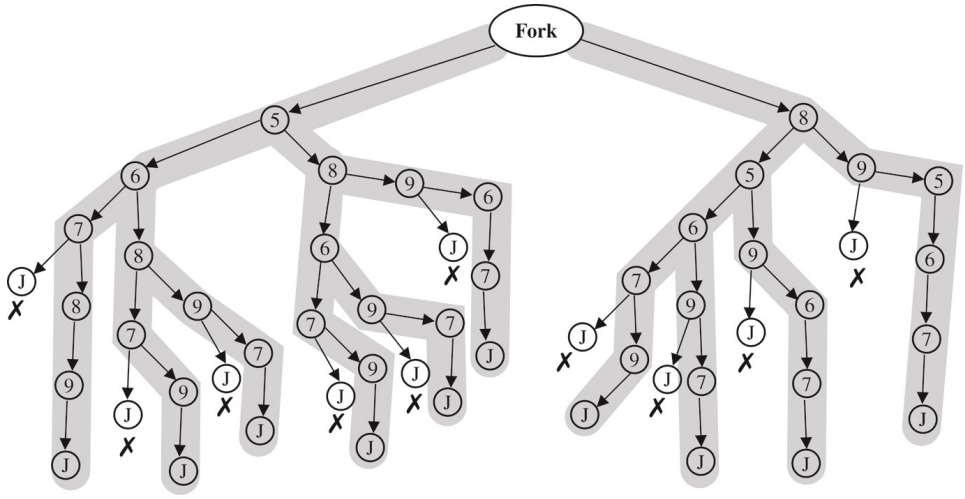


Figure 4.3: Exploratory tree depicting scenarios among existing permutations for the concurrent section in Figure 2.3 of Chapter 2.

The pivotal motivator behind the application of ant colony algorithm for generating the scenarios from a concurrent section of an activity diagram comes from the existing similarity between the pheromone-based path traversal by the ants from their nest to food source, and the test scenarios that might be generated by incremental traversal over the edges present in the similar tree type structure representing the permutations under fork-

join node. Representing a scenario with a single ant nest node (source) and food source node (sink); multiple routes in-between consequently developed into a structure, similar to the one shown in Figure 4.2. Figure 4.3 depicts an exploratory view with all the valid and invalid scenarios generated for the concurrent section taken for Figure 2.3, where ✓ stands for a valid path; and × indicates an invalid path. In Figure 4.3, color variations from dark to light at marked tubular structure represent the path with varying deposition of pheromones, *i.e.* from high deposition to lower one. Higher pheromone deposition leads to redundant traversal of the same path by the ant like agents present in the ant colony.

4.2 Mathematical model for ant system

Ant System (AS) is the first ACO algorithm proposed in the literature [236] and can be applied in various ways. In AS, an ant *k* at node *i* selects the next possible node *j* with probability computed using the random proportional rule, defined as

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha}.[\eta_{ij}]^{\beta}}{\sum_{h \in N^k}[\tau_{ih}]^{\alpha}.[\eta_{ih}]^{\beta}} \quad (1)$$

Where, $\tau_{ij}$is the amount of pheromone on edge (i, j); α is a parameter that controls the influence of τ_{ij}; η_{ij} is termed as the desirability of edge (i, j); β is a parameter which manages the influence of η_{ij}. N^k is the set denoting feasible neighborhood, which excludes nodes already visited in the partial tour of ant *k*; and it may further be restricted to a candidate set of the nearest neighbors of a node *i*. For lowering the impact of heuristic, the probability factor is inversely proportional to the heuristic value. Hence, the parameter that controls the influence of η_{ij} is taken as $(-\beta)$. This changes the Eq. (1) into Eq. (1a) as given below:

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha}.[\eta_{ij}]^{-\beta}}{\sum_{h \in N^k}[\tau_{ih}]^{\alpha}.[\eta_{ih}]^{-\beta}} \quad (1a)$$

The pheromone trail values are updated as

$$\tau_{ij} \leftarrow (1-\rho).\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \quad (2)$$

Where, ρ is the evaporation rate taken for the system; and $\Delta\tau_{ij}^{k}$ is defined as follows:

$$\Delta\tau_{ij}^{k} = \begin{cases} F(k) & if\ edge\ (i,j) is\ a\ part\ of\ the\ solution\ constructed\ by\ ant\ k, \\ 0 & Otherwise \end{cases}$$

Where $F(k)$ is the amount of pheromone deposited on the edges of the solution constructed by ant k. $F(k)$ is reciprocal of the cost of the solution constructed by ant k, possibly multiplied by a constant Q. Hence, better the solution, higher would be the amount of pheromone deposited by an ant [242]. A different version of AS based algorithms has been proposed by various researchers that differ in few parameters like *Construction of Solutions* and procedure for *Update Pheromone* [236].

4.3 Orientation factor and its adaptation

Experimental studies show that ACO is a powerful approach for finding the best solution, but it suffers from long search time and falling into local optimal for large-scale computation. Initially, each agent in ACO moves randomly, and the amount of information is almost same on each route available for traversal. For an effective path, the information symbolizing the amount of pheromone increases gradually through positive feedback.

In Figure 4.4 using simple ACO approach, next hop node is decided by pheromone intensity along the edge between the current node and tentative next node. The issue crops up when initial pheromone intensity along every edge between the current node and the nodes belonging to the set of probable next nodes is the same. Therefore, the ant cannot select the initial edge of the feasible path with bigger probability. An orientation factor has been introduced while computing the probability of selecting the next hop. Orientation-based Ant Colony Optimization (OBACO) considers pheromone intensity as well as *Cosine* of the angular factor (θ) between the current node and the next node.

$$P_{ij}^{'} = P_{ij} \times \frac{1}{Cos\,\theta} \qquad (3)$$

In the case of a large search space, it is cumbersome to locate feasible paths using simple ant colony approach. Hence, by adding an orientation factor in the movement of ants, the count of feasible paths can be increased as compared to simple AS. Repetitive simulations proved that an orientation factor given in Eq. (3) above enhances the count of desired paths for the problem under consideration. The angle θ is orientation between X-axis and the vector taken from $N_{current}$ to the candidate node $N_{nexthop}$. As shown in Figure 4.4 the orientation can be taken between X-axis and the vector directed from the

current node $N_{current}$ to the candidate node $N_{nexthop}$. The orientation is taken either for the node(s) having a parent-child relationship with its predecessor node; or for the one(s) which are not having a direct parent-child relationship, but whose parent node has already been visited.

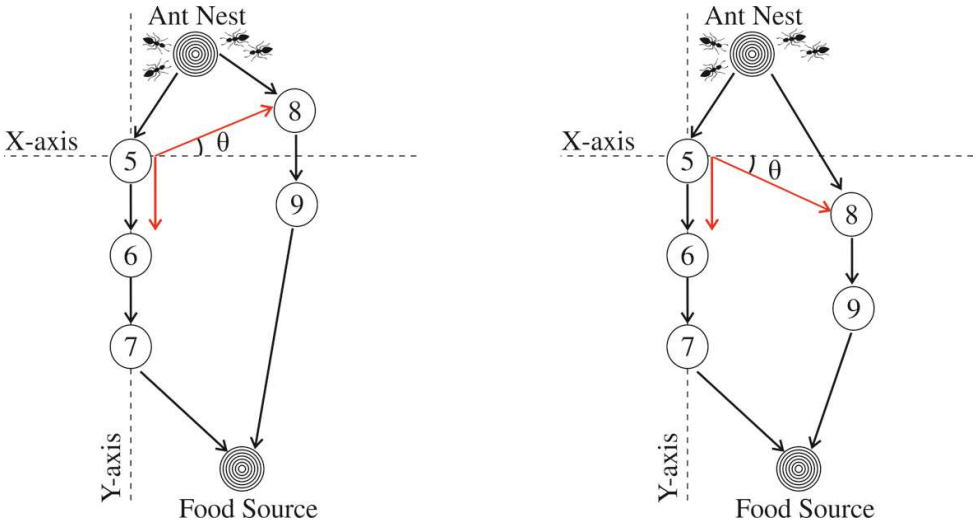


Figure 4.4: Orientation as an influential factor by ant traversal.

The $Cos\ \theta$ is representing the cosine value between the two orientation vectors v_i and the X_{axis}. Using the formula for the dot product as $Cos\ \theta = v_i.X_{axis}/(|v_i|.|X_{axis}|)$, we can compute cosine of angle directly. The orientation factor influences the choice of deciding the next hop rather than falling into local optima. After incorporating the orientation factor, the Eq. (1a) gets converted to Eq. (4) as hereunder:

$$p_{ij}^{'} = \frac{[\tau_{ij}]^{\alpha}.[\eta_{ij}]^{-\beta}}{\sum_{h \in N^k}[\tau_{ih}]^{\alpha}.[\eta_{ih}]^{-\beta}} \qquad (4)$$

The value for p_{ij}, and τ_{ij} are obtained by using Eq. (1a) and (2); and each $p_{ij}^{'}$ is computed by adding an orientation factor, $(Cos\ \theta)^{-1}$ to p_{ij} while selecting the next node from a set of nodes. The present model articulates the evolution process using the variable $p_{ij}^{'}$ that progresses by the adaptation of Eq. (4). Depending on the value of probability, specific edges flourish or persist, while others droop during the ant traversal. It is acknowledged that the system generates test scenarios by selecting the next neighbor node resulting into a bridge between two specific nodes, *i.e.*, $N_1(ant\ nest)$ and $N_2\ (food\ source)$. Here, the modified ant-based approach applies heuristic values and orientation factor to decide for the next node to jump upon. Finally, this approach

generates a path from ant nest node to food source by incremental traversal over each intermediate edge.

The concept of using angular orientation in ant colony optimization algorithm has already been applied for diversified domains that demand exclusivity of computational abilities such as multicast routing in multimedia networks [202], transport network [203], path planning for Unidentified Arial Vehicle (UAV) [204], multi-path routing [243], harmonic elimination [244], probabilistic Traveling Salesman Problem (TSP) [205], *etc*. However, to the best of our knowledge, this orientation-based ant colony optimization approach is yet to be sufficiently researched for imbibing its abilities in solving the problem of test scenario generation for a concurrent section of the UML activity diagram.

4.4 Proposed approach

Firstly, this section lists the steps as shown in Figure 4.5 for getting test scenarios from UML activity diagram; and subsequently, describes the technique followed for the current research study. Activity diagram has been drawn using Rational Software Architect (RSA) and exported to its respective Extensible Markup Language Message Interface (XMI) format.

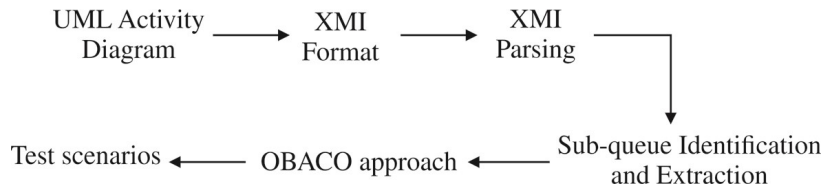


Figure 4.5: Methodology to generate test scenarios from UML activity diagram.

After parsing XMI code, sub-queues present under the fork-join are extracted. The proposed approach using orientation-based ant colony optimization has been applied to generate the permutations among activity nodes of fork-join sub-queues. To develop the proposed approach, some of the main notations used throughout the chapter here, have been defined.

SQ_i	Sub-queues between *fork-join* structure/s of an activity diagram
a_k	k^{th} ant
A_n	Set of n ants

A_n^c	c^{th} set of n ants
$\mathcal{N}_i^c$	Set of neighboring nodes for i^{th} node
p_{ij}^k	Probability of elements present under the set $\mathcal{N}_i^c$
τ_{ij}	Pheromone level at the edge from node i to j
$\Delta\tau_{ij}$	Small change in the value of pheromone for the edge between node i and j
ρ	Evaporation rate of pheromone
η_{ij}	Heuristic value for the edge between node i and j
$Q\ or\ Q''$	Multiplying factor for heuristic value
j_{a_k}	Next node for ant a_k whose probability has been selected from the set p_{ij}^k
α	Factor that controls the pheromone level
β	Factor that controls the heuristic value
d_{ij}	An identifier related to distance between current node (i) and the next node (j)
CSq	Count of sub-queues from fork node
$S_g^{a_k}$	g^{th} sequence for ant a_k
TS_s	Full set of final sequences/scenarios
θ	Orientation between the current node and the next one selected to be jumped upon

4.4.1 Algorithm to generate test scenarios for concurrent section

This sub-section outlines the steps for applying the proposed OBACO approach to compute the test scenario in the concurrent section of UML activity diagram. The algorithm takes fork-join sub-queues (SQ_i) as input. A_n^c is a set of ants belonging to a specific c^{th} colony and having m ants, *i.e.* $A_n^c = \{a_1, a_2, a_3, \ldots\ldots\ldots\ldots a_m\}$. The main iterative construct in the algorithm commences from fork node with a single ant a_k at a time from a particular colony. From the current position a set ($\mathcal{N}_i^c$) of the feasible neighborhood is computed, whose parent node(s) has/have already been visited. The probability value $p_{nj}^k = \{p_{i1}^k, p_{i2}^k, \ldots\ldots\ldots\ldots\ldots p_{id}^k\}$ is computed for all the elements in $\mathcal{N}_i^c$. The next activity node is decided by pheromone, heuristic values, and orientation factor. Here, orientation can be taken between X-axis and the vector directed from the current node $N_{current}$ to the candidate node from $\mathcal{N}_i^c$. After selecting a specific j[th] activity node, the values for pheromone as well as heuristic are updated for the selected one using the formula $\tau_{ij} = \{(1-\rho) \times \tau_{ij}^k\}^{\alpha} + \left\{Q'' \times \left(\frac{1}{d_{ij}}\right)\right\}^{-2\beta}$. The selected j[th] node will be marked for ant a_k and added into a set denoting a particular sequence. Node j will now become the current node as i if j is not a join. After one complete execution of inner $do \ldots While$ loop, one end-to-end sequence between the fork and join node is generated and the same is considered as a test sequence/scenario generated by a particular ant a_k.

Outer for loop gives a set of test sequences (TS_s) as formed by n ants available in a specific colony. Following the above mentioned steps, the proposed algorithm is results into a slice (sub-set) of scenarios that are feasible, by neglecting the infeasible paths.

Input: Sub-queues(SQ_i), where $i = \{1,2,3 \dots n\}$, TN
Output: Test scenario(s)(TS_s): $ts_1, ts_2, \dots ts_m$

Begin

1. **Let** $A_n^c = \{a_1, a_2, a_3, \dots \dots \dots \dots a_m\}$ be a set of ants at fork node for a c^{th}colony Where, c= $\{1, 2, 3, \dots \dots \dots \dots u\}$
2. **for each** $a_k \in A_n^s$ {
3. Initiate the traversal of ant a_k from *fork* node at i^{th} position
4. **do** {
5. Compute the set of elements having j^{th} next node(s) for a_k , where all the parent nodes for the j have already been visited. Denote this set by $\mathcal{N}_i^c$.
6. Compute the probability of elements under $\mathcal{N}_i^c$ using Eq. 1 as given below, where, $p_{ij}^k \neq 0$

$$p_{ij}^k = \frac{[\tau_{ij}]^{\alpha}.[\eta_{ij}]^{-\beta}}{\sum_{l \in \mathcal{N}_i^c}[\tau_{il}]^{\alpha}[\eta_{il}]^{-\beta}} \times \frac{1}{Cos\theta}$$

Here, $j \in \mathcal{N}_i^c$

Where, $\mathcal{N}_i^c$ is the set of feasible neighborhood for k^{th} ant, when being at node i.

$$\theta = \begin{cases} 25^0 \; when\; node\; i\; and\; j\; are\; from\; different\; sub-queues \\ 0^0 \quad Otherwise \end{cases}$$

7. Let $p_{nj}^k = \{p_{i1}^k, p_{i2}^k, \dots \dots \dots \dots \dots p_{id}^k\}$
8. $j_{a_k} =$
$\begin{cases} max.value\; from\; set\; p_{nj}^k \quad if\; there\; exists\; variation\; in\; elements\; of\; set\; p_{nj}^k \\ any\; random\; value\; from\; p_{nj}^k \quad Otherwise \end{cases}$
9. Update the pheromone and heuristic for the edge connecting current node and selected j^{th} node for ant a_k

$$\tau_{ij} = \{(1-\rho) \times \tau_{ij}^k\}^{\alpha} + \{\Delta\tau_{ij}^k\}^{-2\beta}$$

$$\tau_{ij} = \{(1-\rho) \times \tau_{ij}^k\}^{\alpha} + \left\{Q'' \times (\frac{1}{d_{ij}})\right\}^{-2\beta}$$

$$where, Q'' = \begin{cases} \frac{Q}{CSq} & when\; i\; is\; parent\; of\; j\; and\; it\; is\; from\; the\; same\; sub-queue \\ Q & otherwise \end{cases}$$

$$\eta_{ij} = Q'' \times \eta_{ij}^k$$

10. $S_g^{a_k} = S_g^{a_k} \cup \text{j}^{\text{th}}\text{node}$

11. Now, make node j as current node $i \leftarrow j$
12. } (**While** *current node* $i \neq$ *join node* for current *fork-join*)
13. $ts_i \leftarrow S_g^{a_k}$
14. $TS_s = TS_s \cup ts_i$
15. }

End

4.4.2 A sample run of the proposed algorithm

Using a single ant agent, Figure 4.6 (a)-(e) displays the sample run of the proposed approach on the concurrent section of activity diagram as presented in Figure 2.3 (starting from fork node and ending up to join node). In Figure 4.6, co-ordinate axis (X and Y-axis) are represented by the dashed lines.

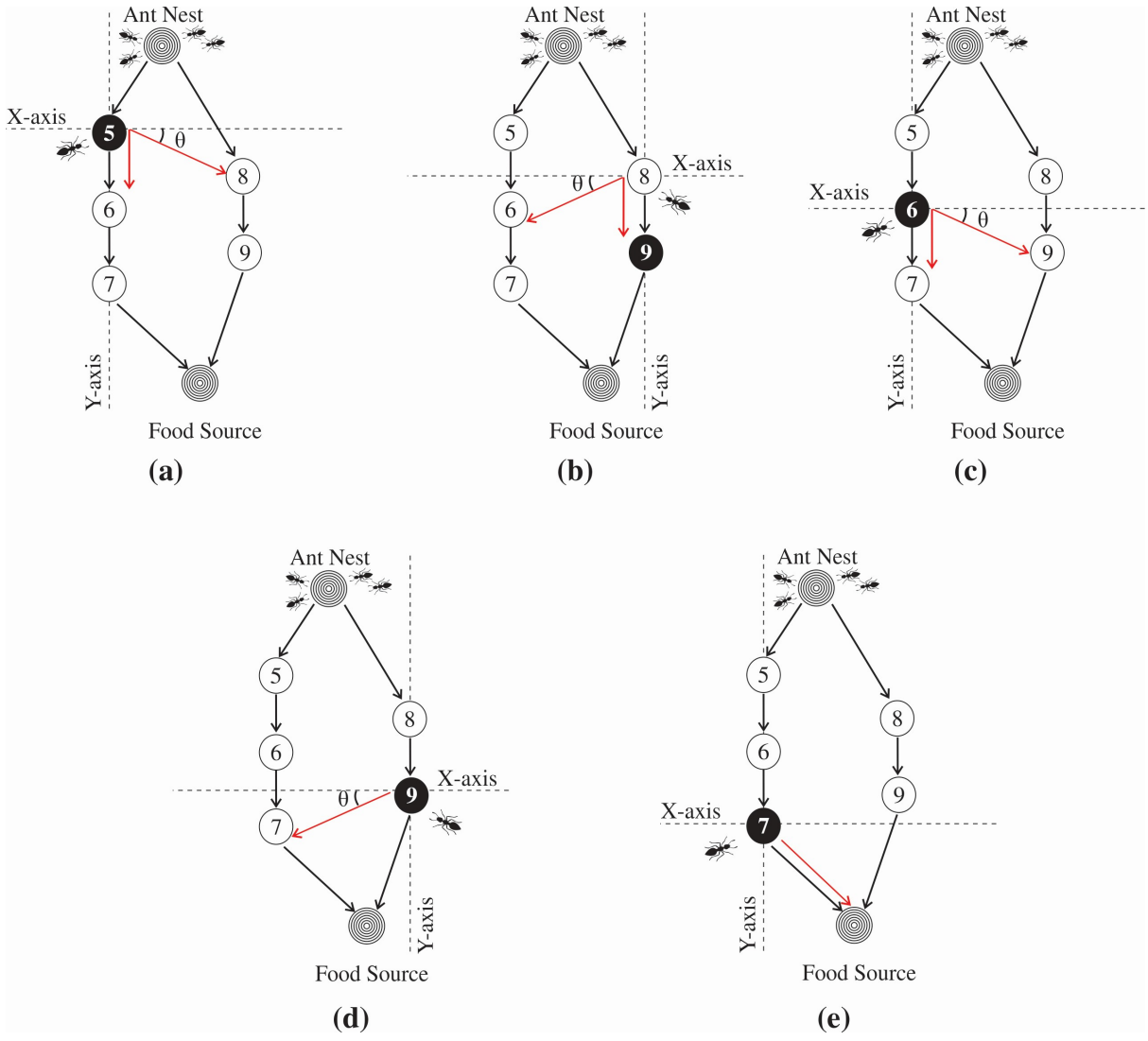


Figure 4.6 (a)-(e): Execution of OBACO using the activity nodes present between fork-join pair

Table 4.1 presents the various parameters and their respective values taken for simulation of the OBACO approach. The labels A, J, and N in column header of Table 4.1 denote current node, Next node to be jumped upon, and Next probable node(s) respectively for an ant agent. The table also shows the intermediate computations involved when an ant agent initiates from fork node and reach up to join node by an incremental traversal over the intermediate activity nodes. An ant $a_k \in A_n^c$ starts its traversal from fork node, where it chooses the next directly connected node at random (either Node 1 or Node 4). Here, let us assume that Ant1 has chosen Node1. From current node, a set constituting feasible neighbors has been computed ($\mathcal{N}_i^c$). The greater value of probability (computed for the set $\mathcal{N}_i^c$) will provide the node on which the control will jump upon. Once the next node has been selected, the value of pheromone as well as heuristic for the path between current node and the selected one has to be increased using the formula $\tau_{ij} = \{(1-\rho) \times \tau_{ij}^k\}^{\alpha} + \left\{Q'' \times (\frac{1}{d_{ij}})\right\}^{-2\beta}$ and $\eta_{ij} = Q'' \times \eta_{ij}^k$.

Table 4.1: Value of various parameters in the proposed ant-based approach

S.No.	Name of the associated parameter	Value
1.	Pheromone τ_{ij}	0.8
2.	Evaporation rate ρ	0.2
3.	Heuristic η_{ij}	2.0
4.	Alpha α	1.1
5.	Beta β	1.0
6.	Constant Q	3.0
7.	Distance factor between two neighbor nodes d_{ij}	1.0
8.	Orientation value θ	25^0 / 0^0

Table 4.2 demonstrates the step-wise and intermediate computations involved for computing next possible node from the current node, where current node is considered to be any among those connected directly to the fork node. This uses the concurrent section of the activity diagram in Figure 2.3.

Table 4.2: Computations involved while performing the traversal of activity nodes from fork to join node using two ant agents (1 and 2) taken from a specific colony

Ant No.	A	N	Computation Involved	J	Update of Pheromone and Heuristic of selected node
1.	5	{6, 8}	$P_{56} = \frac{(\tau_{56})^{1.1} \times (\eta_{56})^{-1}}{[(\tau_{56})^{1.1} \times (\eta_{56})^{-1}] + (\tau_{58})^{1.1} \times (\eta_{58})^{-1}} \times \frac{1}{Cos0^0} = 0.5$ $P_{58} = \frac{(\tau_{58})^{1.1} \times (\eta_{58})^{-1}}{[(\tau_{56})^{1.1} \times (\eta_{56})^{-1}] + (\tau_{58})^{1.1} \times (\eta_{58})^{-1}} \times \frac{1}{Cos25^0} = 0.552$	8	$\tau_{58} = [(1-0.2) \times 0.8]^{1.1} + [3]^{-2(1)} = .732$ $\eta_{58} = 3 \times 2 = 6$
	8	{6, 9}	$P_{86} = \frac{(\tau_{86})^{1.1} \times (\eta_{86})^{-1}}{[(\tau_{86})^{1.1} \times (\eta_{86})^{-1}] + (\tau_{89})^{1.1} \times (\eta_{89})^{-1}} \times \frac{1}{Cos25^0} = 0.552$ $P_{89} = \frac{(\tau_{89})^{1.1} \times (\eta_{89})^{-1}}{[(\tau_{86})^{1.1} \times (\eta_{86})^{-1}] + (\tau_{89})^{1.1} \times (\eta_{89})^{-1}} \times \frac{1}{Cos0^0} = 0.5$	6	$\tau_{86} = [(1-0.2) \times 0.8]^{1.1} + [3]^{-2(1)} = .732$ $\eta_{86} = 3 \times 2 = 6$
	6	{7, 9}	$P_{67} = \frac{(\tau_{67})^{1.1} \times (\eta_{67})^{-1}}{[(\tau_{67})^{1.1} \times (\eta_{67})^{-1}] + (\tau_{69})^{1.1} \times (\eta_{69})^{-1}} \times \frac{1}{Cos0^0} = 0.5$ $P_{69} = \frac{(\tau_{69})^{1.1} \times (\eta_{69})^{-1}}{[(\tau_{67})^{1.1} \times (\eta_{67})^{-1}] + (\tau_{69})^{1.1} \times (\eta_{69})^{-1}} \times \frac{1}{Cos25^0} = 0.552$	9	$\tau_{69} = [(1-0.2) \times 0.8]^{1.1} + [3]^{-2(1)} = .732$ $\eta_{69} = 3 \times 2 = 6$
	9	{7}	$P_{97} = \frac{(\tau_{97})^{1.1} \times (\eta_{97})^{-1}}{[(\tau_{97})^{1.1} \times (\eta_{97})^{-1}] \times \frac{1}{Cos25^0}} = 1.103$	7	$\tau_{97} = [(1-0.2) \times 0.8]^{1.1} + [3]^{-2(1)} = .732$ $\eta_{97} = 3 \times 2 = 6$
	7	**Join**	**Final Sequence as obtained by Ant-1 is** $Fork \rightarrow 5 \rightarrow 8 \rightarrow 6 \rightarrow 9 \rightarrow 7 \rightarrow Join$		
2.	5	{6, 8}	$P_{56} = \frac{(\tau_{56})^{1.1} \times (\eta_{56})^{-1}}{[(\tau_{56})^{1.1} \times (\eta_{56})^{-1}] + (\tau_{14})^{1.1} \times (\eta_{14})^{-1}} \times \frac{1}{Cos0^0} = 0.768$ $P_{58} = \frac{(\tau_{58})^{1.1} \times (\eta_{58})^{-1}}{[(\tau_{56})^{1.1} \times (\eta_{56})^{-1}] + (\tau_{58})^{1.1} \times (\eta_{58})^{-1}} \times \frac{1}{Cos25^0} = 0.256$	6	$\tau_{56} = [(1-0.2) \times 0.8]^{1.1} + [1.5]^{-2(1)} = 1.05$ $\eta_{56} = 1.5 \times 2 = 3$
	6	{7, 8}	$P_{23} = \frac{(\tau_{67})^{1.1} \times (\eta_{67})^{-1}}{[(\tau_{67})^{1.1} \times (\eta_{67})^{-1}] + (\tau_{68})^{1.1} \times (\eta_{68})^{-1}} \times \frac{1}{Cos0^0} = 0.5$ $P_{68} = \frac{(\tau_{68})^{1.1} \times (\eta_{68})^{-1}}{[(\tau_{67})^{1.1} \times (\eta_{67})^{-1}] + (\tau_{68})^{1.1} \times (\eta_{68})^{-1}} \times \frac{1}{Cos25^0} = 0.552$	8	$\tau_{68} = [(1-0.2) \times 0.8]^{1.1} + [3]^{-2(1)} = .732$ $\eta_{68} = 3 \times 2 = 6$
	8	{7, 9}	$P_{87} = \frac{(\tau_{87})^{1.1} \times (\eta_{87})^{-1}}{[(\tau_{87})^{1.1} \times (\eta_{87})^{-1}] + (\tau_{45})^{1.1} \times (\eta_{45})^{-1}} \times \frac{1}{Cos25^0} = 0.552$ $P_{89} = \frac{(\tau_{89})^{1.1} \times (\eta_{89})^{-1}}{[(\tau_{87})^{1.1} \times (\eta_{87})^{-1}] + (\tau_{89})^{1.1} \times (\eta_{89})^{-1}} \times \frac{1}{Cos0^0} = 0.5$	7	$\tau_{87} = [(1-0.2) \times 0.8]^{1.1} + [3]^{-2(1)} = .732$ $\eta_{87} = 3 \times 2 = 6$

Ant No.	A	N	Computation Involved	J	Update of Pheromone and Heuristic of selected node
	7	{9}	$P_{79} = \frac{(\tau_{79})^{1.1} \times (\eta_{79})^{-1}}{[(\tau_{79})^{1.1} \times (\eta_{79})^{-1}]} \times \frac{1}{Cos25^0} = 1.103$	9	$\tau_{79} = [(1-0.2) \times 0.8]^{1.1} + [3]^{-2(1)} = .732$ $\eta_{79} = 3 \times 2 = 6$
	9	Join	**Final Sequence as obtained by Ant-2 is** $Fork \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 7 \rightarrow 9 \rightarrow Join$		

The tube like structure as highlighted in Figure 4.7 shows the paths/sequences obtained in Table 4.2 which use two ant agents (separately) for traversal using the OBACO algorithm. Here, two ant agents resulted into two paths, namely $Fork \rightarrow 5 \rightarrow 8 \rightarrow 6 \rightarrow 9 \rightarrow 7 \rightarrow Join$ and $Fork \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 7 \rightarrow 9 \rightarrow Join$.

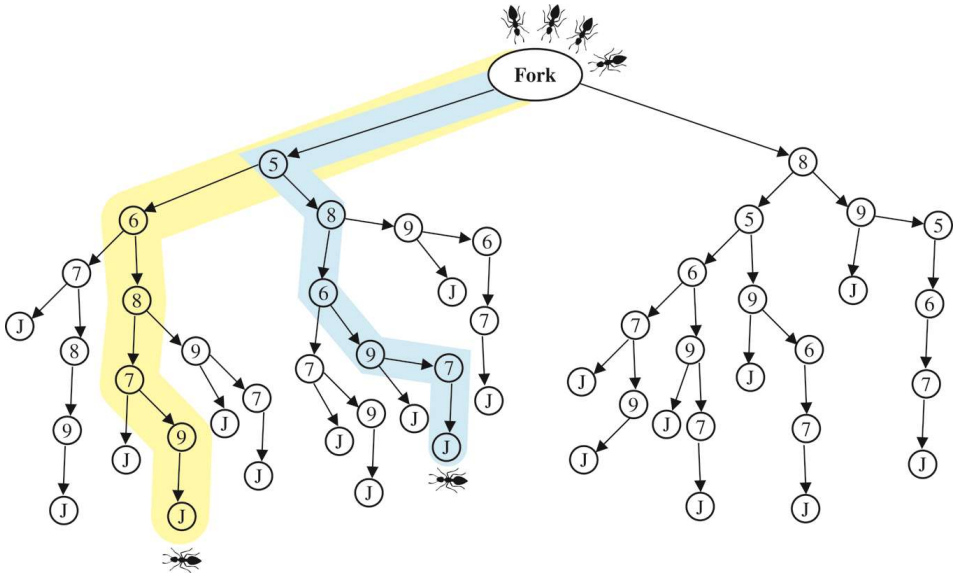


Figure 4.7: Sample run of the proposed approach on an example activity diagram.

4.5 Experimental results and analysis

4.5.1 Parameter setting, stopping criteria and generic settings

To simulate the ACO, GA, and OBACO, the environment and guidelines provided by Črepinšek *et al.* [224] have been considered for performing the experimentation as taken in Sub-section 3.6.1. To perform a comparative analysis of the proposed ant-based approach with already existing GA approach, the control parameters and steps given in Sub-section 3.6.1 have been considered. To simulate ACO, the variant of the ant colony as given by the Srivastava *et al.* [192] has been used. Various parameters with their associated values taken for simulating ACO are as follows: τ_{ij} is the pheromone level at the edge from node i *to* j (1.0), ρ being the evaporation rate of pheromone (0.0), η_{ij} heuristic value for the edge between node i *to* j (2.0), α factor controlling the pheromone

level (1.0), and β factor controlling the heuristic value (1.0). For simulating the OBACO, values for different parameters have been considered as shown in Table 4.1.

For generic settings and the specific stopping criteria, the recommendations listed in Sub-section 3.6.1 have been followed. To simulate the algorithms, stopping criteria has been set as 100000 evaluations for the fitness function. The computational effort of the proposed OBACO has been analyzed by taking into account two factors, namely, (i) Time in milliseconds for the execution of algorithm only; and (ii) Time in milliseconds to convert UML activity diagram into XMI and further up to the execution of the algorithm. In the experimental set-up, where OBACO, GA, and ACO are simulated, the parameter named the number of ant agents in OBACO is considered as equivalent to the population in GA or number of ants in ACO. The number of trials in OBACO and the number of generations in GA or number of trials in ACO has been considered as the same. All the approaches have been executed thirty times for each subject system. For executing the algorithms on C1, the trial count is taken as 10 and population as 100. For comparing the computational effort, the time (in milliseconds) to execute the algorithm only has been considered as the parameter.

4.5.2 Objectives of the experimentation

The OBACO is executed on the subject systems taken from the LINDHOLMEN dataset [231] and synthetic cases as taken in Sub-sections 3.6.3 and 3.6.7 respectively. Student projects considered here are different from those mentioned in Sub-section 3.6.5. The objectives of current experimentation are as follows:

(i) To compare the OBACO with already existing ant colony approach and genetic algorithm on the basis of count generated for feasible test scenarios.
(ii) To conduct statistical analysis for validating the results obtained through the orientation-based ant colony optimization.

4.5.3 Comparison of OBACO with existing GA and ACO

In this sub-section, benchmarks as identified from the LINDHOLMEN dataset have been taken to compare the OBACO with the existing GA and ACO.

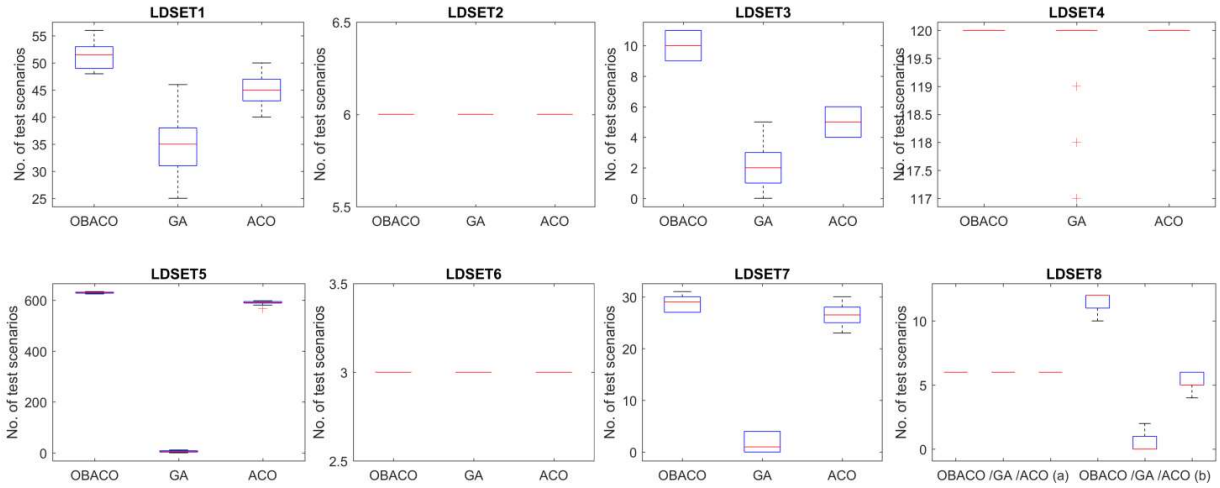


Figure 4.8: Generation of feasible test scenarios using OBACO, GA and ACO.

For the cases taken from LINDHOLMEN dataset, the box-plot shown in Figure 4.8 depicts that OBACO is providing more number of feasible test scenarios as compared to GA and ACO for LDSET1, LDSET3, LDSET5, and LDSET8b. For the benchmarks, LDSET2, LDSET4, LDSET6 and LDSET8a, OBACO has provided results equivalent to GA and ACO. For LDSET7, OBACO has given results equivalent to ACO, but better than GA.

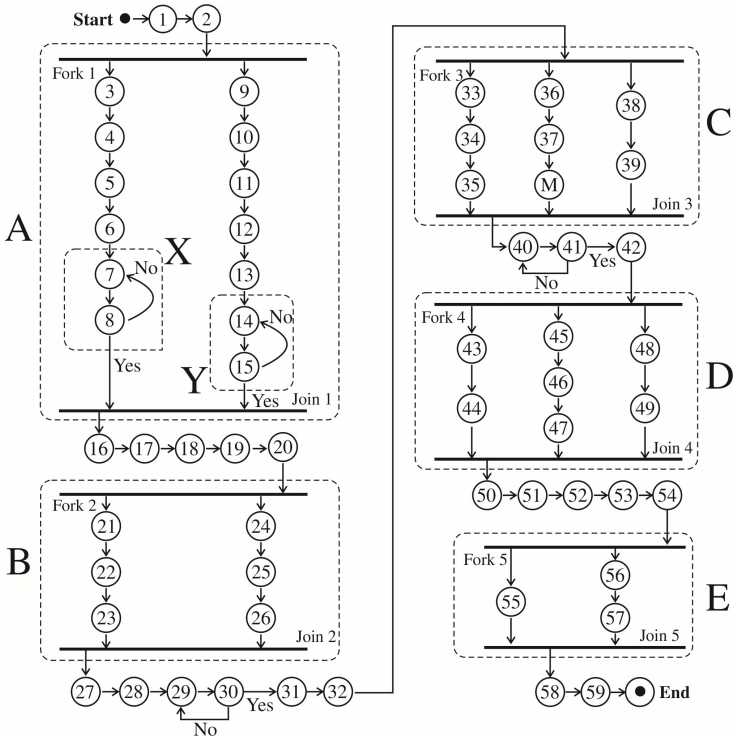
4.5.4 Student project as subject system

Following the similar strategy as mentioned in Sub-section 3.6.5, two student projects have been taken for simulating the proposed approach and compared with GA and ACO. Table 4.3 describes the size measures for the two activity diagram models created by the student groups. Here, the count of elements under fork-join is approximately three times as compared to almost all the cases of LINDHOLMEN dataset. Hence, the models under student project also served a great purpose to test the scalability of the proposed approach.

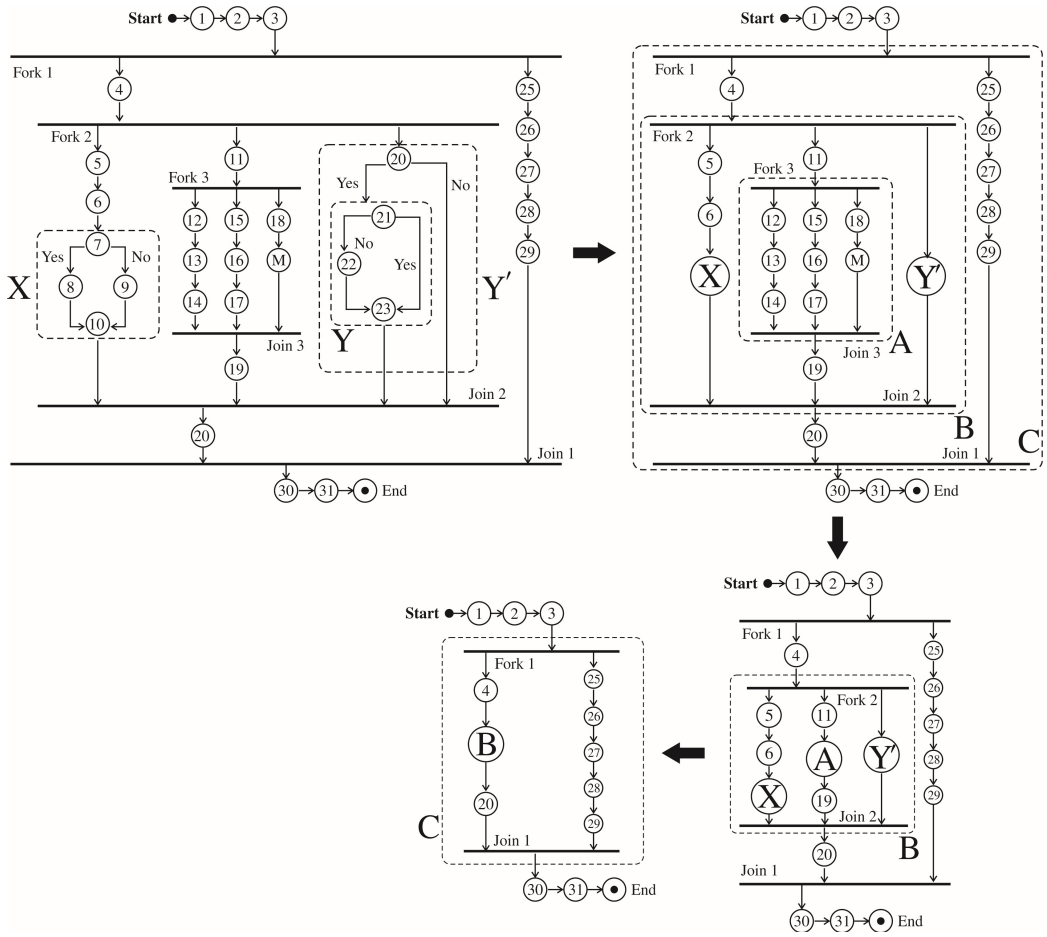
Table 4.3: Size measures of the activity diagrams with number of total elements.

Model No.	AD Model [232]	No. of students involved	Count of elements in AD	XMI File (KB)	Count of fork-join(s) in AD	Count of control construct(s) in AD	Nested Fork-Join
STDP1	**SDLC**	3	154	45	5	4	No
STDP2	**WebCrawling**	5	87	26	3	3	Yes

All the activity diagrams with nested fork-join structure and control constructs like the loop, if-else, *etc.* have been transformed first into an Intermediate Testable Model (ITM) that serves as an input activity diagram with no nested/control construct [190]. For attaining the desired simplified activity diagram, the approach used in Sub-section 3.6.5 has been followed. Here, the proposed OBACO has been applied to each independent and simplified activity diagram having concurrency construct (with no nested or control constructs) for generating the test scenarios. Figure 4.9 highlights the whole transformational process of receiving a simplified activity diagram from the complex nested version of the same, and it serves as an input for the experimentation.



(a)



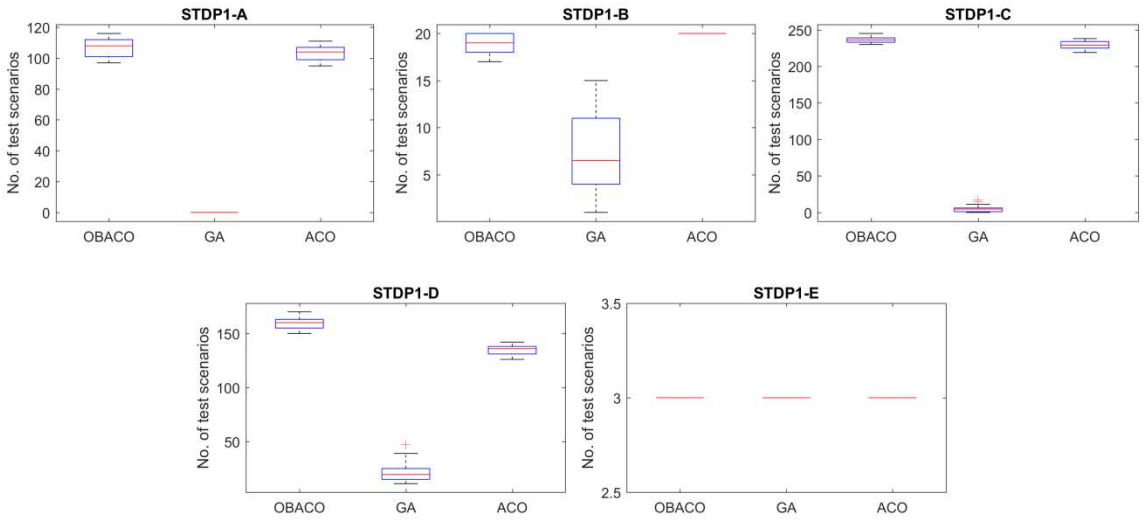
(b)

Figure 4.9 (a) and (b): Transformational process for the activity diagram titled SDLC and web crawling respectively.

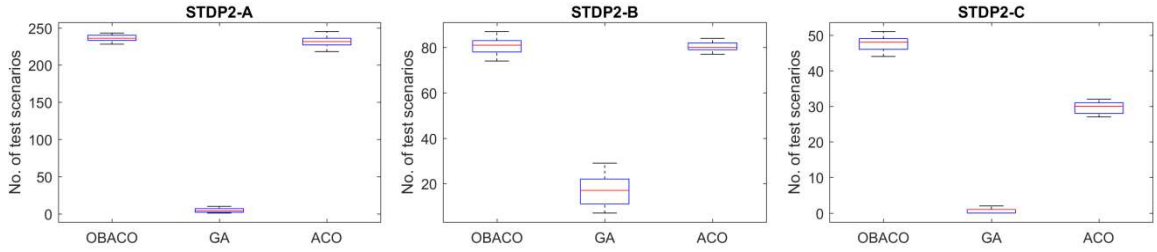
Table 4.4 shows the count of scenarios that can be generated using the formula in [189] for each sub-section of the graph shown in Figure 4.9 for STDP1 and STDP2.

Table 4.4: Count of test scenarios generated using the formula in [189].

Model No.	Model Name	Annotation for the sub-section in Figure 4.9	Count of scenarios generated using [189]
STDP1	SDLC	A	462
		B	20
		C	560
		D	210
		E	3
STDP2	WebCrawling	A	560
		B	140
		C	56



(a)



(b)

Figure 4.10: Count of feasible test scenarios generated using OBACO, GA and ACO on (a) student project 1 (STDP1); and (b) student project 2 (STDP2).

The box-plot as shown in Figure 4.10 (a) and (b) explains that OBACO provides either more or equal number of feasible test scenarios as compared to its meta-heuristic peers, GA and ACO for the sub-activity diagrams (A, B, C, D, E for student project 1; and A, B, C for student project 2) of Figure 4.9. The application of OBACO on the final simplified version of activity diagram has resulted into generating feasible test scenarios with a lot of annotated nodes (each representing nested fork-join or control construct). OBACO is recursively applied to generate further feasible test scenarios on annotated nodes representing nested fork-join. Annotated nodes representing control constructs get substituted by considering its all simplified paths from start to the end node [190]. By doing the substitution for annotated nodes, the final count of feasible test scenarios increases greatly.

For the student projects considered here, GA is performing poor because of inherent randomness at crossover and mutation phase in the said algorithm. The experiment shows that when node counts in the sub-queues under fork-join increases, the

randomness factor in the algorithm will lead to an erratic combination of the nodes in the final scenarios.

4.5.5 Activity diagram with multiple join nodes

In this sub-section, the synthetic cases for activity diagrams with one fork node and multiple join nodes have been taken from Sub-section 3.6.7. Box-plot, as given in Figure 4.11, presents the results of executing OBACO, GA, and ACO on synthetic activity diagrams.

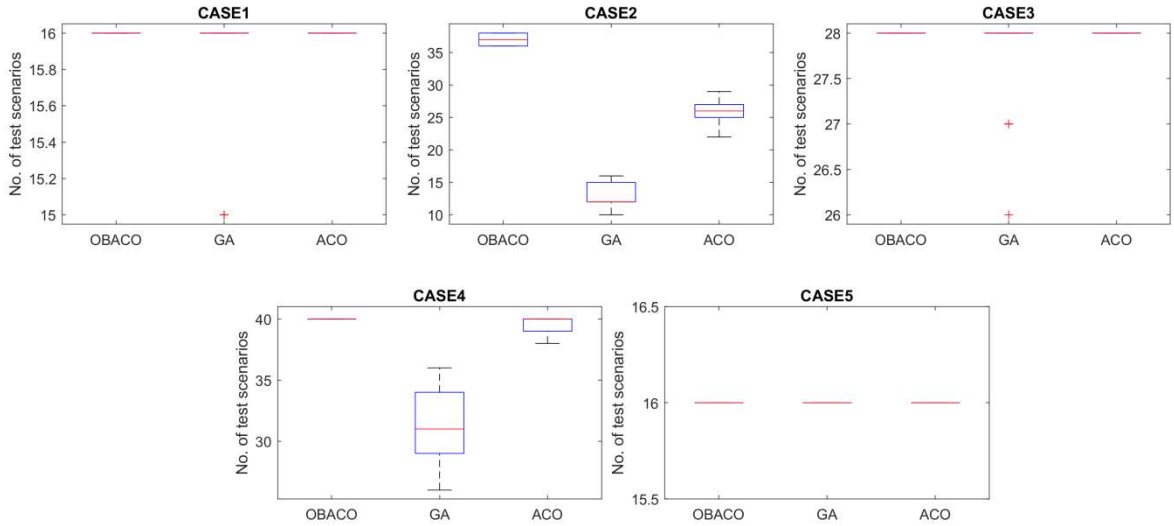


Figure 4.11: Generation of feasible test scenarios using OBACO, GA and ACO.

The results shown in Figure 4.11 indicate that the OBACO is either better or equivalent to GA and ACO about the number of feasible test scenarios obtained.

4.6 Statistical Analysis

Using the same methodology, significance level and set of parameters as presented in Section 3.7, the t-test was applied to get the OBACO, ACO and GA results. Tables 4.5 and 4.6 display the t-test results taking into account the feasible test scenarios as a parameter for OBACO, GA, and ACO.

Table 4.5: Results of t-test taking into account feasible test scenarios generated as a parameter for OBACO and GA.

Model No.		OBACO Vs GA					
		h	**p**	**SD OBACO**	**SD GA**	**Mean OBACO**	**Mean GA**
LDSET1		1	5.51E-22	**2.129432497**	5.59802921	**51.5**	34.8
LDSET2		NaN	NaN	0	0	6	6
LDSET3		1	1.09E-34	**0.803011573**	1.4367908	**10.1**	1.93333333
LDSET4		1	0.011041421	0	0.7648905	**120**	119.633333
LDSET5		1	1.58E-116	**3.280909602**	3.37826045	**630.1666667**	5.36666667
LDSET6		NaN	NaN	0	0	3	3
LDSET7		1	7.75E-57	**1.510499651**	1.60781139	**28.83333333**	1.63333333
LDSET8	a	NaN	NaN	0	0	6	6
	b	1	1.72E-51	**0.758098044**	0.77385436	**11.33333333**	0.56666667

Table 4.6: Results of t-test taking into account feasible test scenarios generated as a parameter for OBACO and ACO.

Model No.		OBACO Vs ACO					
		h	**P**	**SD OBACO**	**SD ACO**	**Mean OBACO**	**Mean ACO**
LDSET1		1	2.38E-14	**2.129432497**	2.59087701	**51.5**	45.3333333
LDSET2		NaN	NaN	0	0	6	6
LDSET3		1	1.56E-31	**0.803011573**	0.81930725	**10.1**	5.13333333
LDSET4		NaN	NaN	0	0	120	120
LDSET5		1	9.33E-39	**3.280909602**	5.84473434	**630.1666667**	590.666667
LDSET6		NaN	NaN	0	0	3	3
LDSET7		1	1.58E-06	**1.510499651**	2.02541325	**28.83333333**	26.3666667
LDSET8	a	NaN	NaN	0	0	6	6
	b	1	2.60E-37	**0.758098044**	0.8051558	**11.33333333**	5.2

The test has been carried out antagonistically towards the null hypothesis that there exists no distinction in their population means. Other than NaN, the value of h as **one** denotes rejection of the null hypothesis. In each case, the p-values obtained are less than the significance level $\alpha = 0.05$. Hence, there exists a difference between the two types of means. In addition to it, the average of the test scenarios obtained for the proposed OBACO approach is better; and standard deviation (SD) for the results obtained using OBACO is less than the existing ACO and GA. Thus, the results are statistically significant.

4.7 Threats to Validity

This section discusses the prospective threats about the validity of subject system used. The potential threats taken up for consideration are as follows:

- The threats to **construct validity** deal with the issue of measurement and raise concerns for measuring the things that have to be measured and ignored in no case. A failure in this regard will raise serious threats to validity. First threat to construct validity is where test sequence generation techniques involve pheromone-heuristic (PH) value calculations. PH values play an important role in deciding the next node in the sequence, and hence, are the vital deciding factor for full sequence generation. Some static initial values have been used for the PH constructs. Whereas, some other estimates may turn to be of more precise significance and enhance the efficiency of pheromone-heuristic centered approaches. Second, although the allied factor related to orientation has been taken from parameter sensitivity analysis and the corresponding value is taken to be a whole number, but some floating point values may enhance the efficiency of the proposed OBACO.

- **Internal validity** is concerned with the performance analysis of meta-heuristic search techniques having biased selection of datasets that can favor a certain technique. The experiments have been conducted using the student projects and dataset available publicly at Github repository. Here, probably a more significant number of datasets seem necessary, as real world datasets are considered confidential by the companies that own them. At this time, the effects related to the locality of model changes (*i.e.* cross synchronization, automatic detection of missing nodes, *etc.*) are not considered.

- **External validity** deals with the generalization of observed results. It causes a major threat to validity. Higher the complexity of a real world UML activity diagram more would be its generalization. But UML diagrams related to real world software projects are difficult to obtain due to the privacy policies followed by software development companies. As a result, it is quite difficult to assess the

extent to which the proposed OBACO algorithm is helpful to the software developers, testers and analysts. However, it is believed that to generate the scenarios for development as well as testing the proposed technique is considerably helpful.

4.8 Conclusion

While concluding, it can be said that experimental evaluation of OBACO, GA, and ACO on UML activity diagram reveal that the main concern arises when the sub-queue(s) under fork-join pair interferes with the execution of respective adjacent sub-queue(s) and creates a sequence explosion. The analysis shows that superior solutions for the analogous computational power are recognized. At present, the studies of ant colony algorithm for generating test sequences are preeminent to researchers; and further study of heuristic techniques is vital to determine their cost-effectiveness. They can be empowered for higher gains through orientation-based ant colony algorithm. The OBACO approach is a good alternative as compared to GA and ACO for finding the feasible set of paths when elements in the set of total paths are opulent.

Chapter 5

Conclusion and Scope for Further Research

5.1 Conclusion and Application potentials

Test scenario/sequence generation through the use of UML activity diagram model has been the prime area of this research work. In the context of test-based software development, especially the software testing, model-based testing is significant as scenarios can be developed automatically by using software models. To identify all the possible scenarios and find the faults in scenarios of a use case, UML behavioral diagram is taken as a useful design construct. This research undertakes the issue of generating test scenarios from UML activity diagrams. The existing work on test scenario generation from UML activity diagram suffers from redundancy in the case of simple ACO, generation of diminished count of test scenarios in the case of GA, and more waiting time in the case of exact algorithms like DFS. It is necessary to limit the gap between models and the systems for which the models have been devised. This is desirable to cope with the complication of present-day software. In other words, available techniques have limitations in their applicability to large and complex UML design models. This work proposes two meta-heuristic approaches which can be applied in modified form for generating scenarios from the fork-join construct in UML activity diagram. Detailed experiments on LINDHOLMEN dataset, student projects, and synthetic activity diagrams reveal that the proposed meta-heuristic approaches are better as compared to the existing bio-inspired techniques for generating the test scenarios in the concurrent section of UML activity diagram. XMI files corresponding to UML activity diagrams taken as input have been used for both these approaches. XMIs have been parsed for fetching the sub-queues present under fork-join nodes. Specific heuristic factors have been used in both the techniques for generating the test scenarios starting from fork node and ending at join node.

The results of present research can be extended to other fields in the following ways:

1. The proposed approach can be directly applied to sequence diagrams. A sequence diagram is closely related to a UML activity diagram and can be converted into it easily. A sequence diagram represents dynamic aspects of an approach/framework under consideration through messages and the replies related to the collaborating entities. In other words, a sequence diagram can explicitly model the constructs like parameters, method calls, collaborating objects and the return values. This results into the specifications that capture the flow of system, and also provides the inputs for generating test scenarios. These test scenarios can precisely check the insufficiencies related to requirement specifications, software development pattern and implementation faults at early stage of SDLC. Similarly, the proposed approach can be applied to interaction-based models communication diagrams, timing diagrams, *etc*.

2. In literature, various *visualization* techniques exist to depict the paths in a graph construct. Proposed techniques can be used for visualizing the paths in complex and large activity diagrams, where comprehension of the control flow is considered as a tough task. Thus, the proposed approach of representing scenarios in a concurrent section of the activity diagrams can be considered as an effective *visualization* technique.

5.2 Scope for further research

Further studies may explore the test scenario selection techniques. Right now, only positive test scenarios are generated, where expected outputs are according to the original software requirement specifications documentation. A test scenario would be a negative test scenario, which declares that the interactions inside the fork-join fragments are not to be executed. In the context of test synthesis, the generation of test scenario can be explored further. However, the cost of test-based development and testing is related to the count of test cases generated. Hence, it is required to filter out a subset from all possible test scenarios for the phase related to the scenario and (or) test execution. To curtail the initial set of scenarios, the impact of interleaving sequences among activity nodes needs to be analyzed. In this work, concurrent coverage criterion considers all potential interleaving sequences among activity nodes under fork-join pair. As a result, the count of test scenarios generated is too large since the count of possible interleaving

sequences enhances exponentially with the count of activity nodes in the sub-queues under fork-join. Here, operations performed on the objects or the activity nodes can be taken in conjunction with read and write operations to analyze the possibility of identifying the occurrences of irrelevant sequences. Following this, the count of valid interleaving sequences, and finally, the count of test cases can be drastically reduced.

Further research may investigate the implementation of the proposed amoeboid organism-based algorithm in various other ways. Firstly, the proposed approach can be merged with parallel BAT algorithm to automatically select the best suitable scenarios having maximum coverage. Secondly, artificial bee colony (ABC) algorithm can be applied with AOA to generate the test case from test scenarios, where values for the variable used in test scenario can be determined automatically. Thirdly, AOA can be applied in a parallel computing environment to facilitate the computation of sequences under fork-join pair more efficiently for effective model-based testing. The proposed orientation-based ant colony algorithm may further investigate the research by implementing it in different directions. Firstly, the proposed approach can be explored for making clusters of test sequences as generated from UML activity diagram to devise a cost-effective approach for regression testing, which executes a specific cluster after having a change in the requirement specifications of the software. Secondly, in conjunction with software re-modularization, OBACO can be applied to group the test sequences in the context of coupling and cohesion for better structural testing.

In spite of certain limitations, the proposed amoeboid-based and orientation-based ant-colony optimization meta-heuristic algorithms may trigger the present technological research by providing abundant of model-based software testing prospects for attaining academic excellence and leveraging industrial pull.

Bibliography

[1] M. Walicki, and S. Meldal, “Nondeterminism vs. Underspecification,” in Multi-Conference on Systemics, Cybernetics and Informatics, Orlando, Florida, USA, 2001, pp. 551-555.

[2] V. Arora, R. Bhatia, and M. Singh, “A systematic review of approaches for testing concurrent programs,” *Concurrency and Computation: Practice and Experience,* vol. 28, no. 5, pp. 1572-1611, 2016.

[3] D. M. Dhamdhere, *Operating Systems A Concept-Based Approach*, 2nd ed.: The Tata McGraw-Hill.

[4] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*: Morgan & Claypool, 2012.

[5] S. Dalai, A. A. Acharya, and D. P. Mohapatra, “Test Case Generation For Concurrent Object-Oriented Systems Using Combinational UML Models,” *International Journal of Advanced Computer Science and Applications,* vol. 3, no. 5, pp. 95-100, 2011.

[6] P. Parashar, A. Kalia, and R. Bhatia, “How Time-Fault Ratio helps in Test Case Prioritization for Regression Testing,” *International Journal of software Engineering (IJSE),* vol. 5, no. 2, pp. 25-35, 2012.

[7] M. Friske, and B.-H. Schlingloff, “Improving test coverage for UML state machines using transition instrumentation,” in International Conference on Computer Safety, Reliability, and Security, 2007, pp. 301-314.

[8] M. Aggarwal, and S. Sabharwal, “Test case generation from UML state machine diagram: A survey,” in IEEE 3rd International Conference on Computer and Communication Technology (ICCCT), 2012, pp. 133-140.

[9] "OMG certified UML professional-2, Concurrency in UML," 22nd June, 2016; http://www.omg.org/ocup-2/documents/concurrency_in_uml_version_2.6.pdf.

[10] M. Khandai, A. A. Acharya, and D. P. Mohapatra, “A novel approach of test case generation for concurrent systems using UML Sequence Diagram,” in IEEE 3rd International Conference on Electronics Computer Technology (ICECT), 2011, pp. 157-161.

[11] W. Rhmann, and V. Saxena, “Test Cases Generation for Object-oriented Software from Decision Slicing of UML Activity Diagram,” *Journal of Scientific Research & Reports,* vol. 11, no. 5, pp. 1-10, 2016.

[12] "Behavioral Modeling with States and Transitions," 22nd June, 2016; http://www.site.uottawa.ca/~bochmann/SEG-2106-2506/Notes/M1-2-StateMachines/index.html.

[13] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, “Model-driven development using UML 2.0: promises and pitfalls,” *Computer,* vol. 39, no. 2, pp. 59-66, 2006.

[14] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand, *Model transformations as a strategy to automate model-based testing-A tool and industrial case studies,* 2010-01, Simula Research Laboratory, 2010.

[15] M. Shirole, and R. Kumar, “Testing for Concurrency in UML Diagrams,” *ACM SIGSOFT Software Engineering Notes,* vol. 37, no. 5, pp. 1-8, 2012.

[16] D. Xu, H. Li, and C. P. Lam, “Using adaptive agents to automatically generate test scenarios from the UML activity diagrams,” in 12th Asia-Pacific Software Engineering Conference, 2005, pp. 8.

[17] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed., New York, NY: McGraw-Hill 2010.

[18] P. Nanda, D. P. Mohapatra, and S. K. Swain, “Generation of Test Scenarios Using Activity Diagram,” in SPIT-IEEE Colloquium and International Conference, Mumbai, India, 2008, pp. 69-73.

[19] M. Young, R. N. Taylor, D. L. Levine, K. A. Nies, and D. Brodbeck, “A concurrency analysis tool suite for Ada programs: Rationale, design, and preliminary experience,” *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 4, no. 1, pp. 65-106, 1995.

[20] R. E. Nance, “The time and state relationships in simulation modeling,” *Communications of the ACM,* vol. 24, no. 4, pp. 173-179, 1981.

[21] B. P. Zeigler, “System-theoretic representation of simulation models,” *IIE Transactions,* vol. 16, no. 1, pp. 19-34, 1984.

[22] D. L. Parnas, “Really rethinking'formal methods,” *Computer,* vol. 43, no. 1, pp. 28-34, 2010.

[23] B. P. Zeigler, *Integrated model pluralism: An alternative to a universal modeling language*, Department of Applied Mathematics, The Weizmann Institute of Science, , Rehovot, Israel, 1979.

[24] A. De, A. Awasthi, and M. Kumar Tiwari, "Robust formulation for optimizing sustainable ship routing and scheduling problem," *IFAC-PapersOnLine,* vol. 48, no. 3, pp. 368-373, 2015.

[25] M. Harman, and B. F. Jones, "Software engineering using metaheuristic innovative algorithms: workshop report," *Information and Software Technology,* vol. 43, no. 14, pp. 905-907, 2001.

[26] Y. Zhang, J. Sun, and S. Jiang, "An application of program slicing in evolutionary testing," in International Conference on Information Engineering and Computer Science (ICIECS), Wuhan, China, 2009, pp. 1-4.

[27] T. Jiang, N. Gold, M. Harman, and Z. Li, "Locating dependence structures using search-based slicing," *Information and Software Technology,* vol. 50, no. 12, pp. 1189-1209, 2008.

[28] G. H. Hwang, K. C. Tai, and T. L. Huang, "Reachability Testing: An Approach to Testing Concurrent Software," in First Asia-Pacific Software Engineering Conference, Tokyo, Japan, 1994, pp. 246-255.

[29] R. H. Carver, and K. C. Tai, "Replay and Testing for concurrent programs," *IEEE Software,* vol. 8, no. 2, pp. 66-74, 1991.

[30] R. N. Taylor, "A general purpose algorithm for analyzing concurrent programs," *Communications of the ACM,* vol. 26, no. 5, pp. 361-376, 1983.

[31] R. Carver, and K. C. Tai, "Deterministic execution testing of concurrent Ada programs," in Tri-Ada '89: Ada technology in context:application, development, and deployment, Pittsburgh, Pennsylvania, United States, 1989, pp. 528-544.

[32] K. C. Tai, R. H. Carver, and E. E. Obaid, "Debugging concurrent Ada program by deterministic execution," *IEEE Transactions on Software Engineering,* vol. 17, no. 1, pp. 45-63, 1991.

[33] K. C. Tai, "Reachability Testing of Asynchronous Message-Passing Programs," in Second International Workshop on Software Engineering for Parallel and Distributed Systems, Boston, MA, 1997, pp. 50-61.

[34] Y. Lei, and K. C. Tai, "Efficient Reachability Testing of Asynchronous Message-Passing Programs," in Proceedings of the Eighth IEEE International

Conference on Engineering of Complex Computer Systems (ICECCS), Greenbelt, MD, USA, 2002, pp. 35-44.

[35] C. Harvey, and P. Strooper, "Testing Java Monitors through Deterministic execution," in Proceedings of the Australian Software Engineering Conference (ASWEC), Canberra, ACT, 2001, pp. 61-67.

[36] P. B. Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* vol. 1, no. 2, pp. 199-207, 1975.

[37] S. Iyer, and S. Ramesh, "Apportioning: A technique for efficient reachability analysis of concurrent object-oriented programs," *IEEE Transactions on Software Engineering,* vol. 27, no. 11, pp. 1037-1056, 2001.

[38] Y. Lei, and R. H. Carver, "Reachability testing of semaphore-based programs," in Proceeding of the 28th Annual International Computer Software and Applications Conference (COMPSAC), Hong Kong, 2004, pp. 312-317.

[39] Y. Lei, and R. H. Carver, "A New Algorithm for Reachability Testing of Concurrent Programs," in Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE), Taipei, Taiwan, 2005, pp. 346-355.

[40] Y. Lei, and R. H. Carver, "Reachability Testing of Concurrent Programs," *IEEE Transactions on Software Engineering,* vol. 32, no. 6, pp. 382-403, 2006.

[41] R. H. Carver, and Y. Lei, "Distributed reachability testing of concurrent programs," *Concurrency and Computation: Practice and Experience,* vol. 22, no. 18, pp. 2445–2466, 2010.

[42] S. R. S. Souza, P. S. L. Souza, M. C. C. Machado, M. S. Camillo, A. Simao, and E. Zaluska, "Using Coverage and Reachability Testting to improve Concurrent Program testing quality," in Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE), Eden Roc Renaissance, Miami Beach, USA, 2011, pp. 207-212.

[43] L. U. Chao, and L. U. Yansheng, "Combination Reduction: A SYN-Sequence Selection Strategy for Reachability Testing of concurrent Programs," *Wuhan University Journal of Natural Sciences,* vol. 12, no. 6, pp. 1024-1028, 2007.

[44] P. U. Fangli, and L. U. Yansheng, "SYN-Sequence Selection Strategy for Testing Concurrent Programs Based on little strong happened-before," *Wuhan University Journal of Natural Sciences,* vol. 14, no. 4, pp. 317-320, 2009.

[45] G. H. Hwang, H. Y. Lin, S. Y. Lin, and C. S. Lin, "Statement Coverage Testing for Nondeterministic Concurrent Programs," in Proceedings of the Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE), Beijing, 2012, pp. 263-266.

[46] S. Q. Li, H. Y. Chen, and Y. X. Sun, "A Framework of Reachability Testing for Java Multithreaded Programs," in Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (Vol.3) (SMC), Netherlands, 2004, pp. 2730–2734.

[47] R. H. Carver, and Y. Lei, "A General Model for Reachability Testing of Concurrent Programs," *Lecture Notes in Computer Science,* vol. 3308, pp. 76-98, 2004.

[48] X. Gong, Y. Wang, Y. Zhou, and B. Li, "On Testing Multithreaded Java Programs," in Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (Vol. 1) (SNPD), Qingdao, China, 2007, pp. 702-706.

[49] R. H. Carver, and J. Lei, "A Stateful approach to Testing Monitors in Multithreaded Programs," in Proceedings of the IEEE 12th International symposium on High Assurance Systems Engineering (HASE), San Jose, CA, 2010, pp. 54-63.

[50] F. Pu, and H. Y. Xu, "A feasibility Strategy for Reachability testing of Internet-Based Concurrent Programs," in Proceedings of the IEEE International Conference on Networking, Sensing and Control (ICNSC), Sanya, 2008, pp. 1559-1564.

[51] R. H. Carver, and Y. Lei, "A class library for implementing, testing and debugging concurrent programs," *International Journal on Software Tools for Technology Transfer,* vol. 12, no. 1, pp. 69-88, 2010.

[52] R. H. Carver, and K. C. Tai, *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*: Wiley, 2005.

[53] C. S. Lin, and G. H. Hwang, "State-cover testing for nondeterministic terminating concurrent programs with an infinite number of synchronization sequences," *Science of Computer Programming,* vol. 78, no. 9, pp. 1294-1323, 2013.

[54] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," *IEEE Transaction on Software Engineering,* vol. 18, no. 3, pp. 206-215, 1992.

[55] M. J. Harrold, and B. A. Malloy, "Data flow testing of parallelized code," in Proceerdings of Conference on Software Maintenance, Orlando, FL, 1992, pp. 272-281.

[56] R. D. Yang, and C. G. Chung, "A Path Analysis approach to concurrent program testing," in Proceedings of the Ninth Annual International Phoenix Conference on Computers and Communications, Scottsdale, AZ, 1990, pp. 425-432.

[57] R. D. Yang, and C. G. Chung, "Path analysis testing of concurrent programs," *Information and Software Technology,* vol. 34, no. 1, pp. 43-56, 1992.

[58] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang, "BPEL4WS Unit Testing: Test case generation using a concurrent path analysis approach," in Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE), Raleigh, NC, 2006, pp. 75-84.

[59] M. A. S. Brito, S. R. S. Souza, and P. S. L. Souza, "An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs," *Procedia Computer Science* vol. 18, pp. 250-259, 2013.

[60] C. S. D. Yang, and L. L. Pollock, "An algorithm for All-du-path testing coverage of shared memory parallel programs," in Proceedings of the Sixth Asian Test Symposium (ATS), Akita, 1997, pp. 263-268.

[61] C. S. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," *ACM SIGSOFT Software Engineering Notes,* vol. 23, no. 2, pp. 153-162, 1998.

[62] H. Seidl, and B. Steffen, "Constraint-based Inter-procedural analysis of parallel programs," *Lecture Notes in Computer Science,* vol. 1782, pp. 351-365, 2000.

[63] P. V. Koppol, R. H. Carver, and K. C. Tai, "Incremental Integration Testing of Concurrent Programs," *IEEE Transaction on Software Engineering,* vol. 28, no. 6, pp. 607-623, 2002.

[64] P. V. Koppol, and K. C. Tai, "An Incremental approach to structural testing of concurrent software," in Proceedings of the ACM SIGSOFT International symposium on Software testing and analysis (ISSTA), San Diego, California, United States, 1996, pp. 14-23.

[65] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha, "TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs " *Lecture Notes in Computer Science,* vol. 7273, pp. 219-234, 2012.

[66] G. Cantone, and A. Esposito, "D-Graphs for structural testing of concurrent and communicating Ada tasks," in Proceedings of the Second International Conference on Software Engineering for Real Time Systems, Cirencester, 1989, pp. 80-84.

[67] F. S. Sarmanho, P. S. L. Souza, S. R. S. Souza, and A. S. Simao, "Structural testing for semaphore-based multithreaded programs," *Lecture Notes in Computer Science,* vol. 5101, pp. 337-346, 2008.

[68] W. Wong, Y. Lei, and X. Ma, "Effective generation of test sequences for structural testing of concurrent programs," in Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Shanghai, China, 2005, pp. 539-548.

[69] D. Grunwald, and H. Srinivasan, "Data flow equations for explicitly parallel programs," *ACM SIGPLAN Notices,* vol. 28, no. 7, pp. 159-168, 1993.

[70] J. Takahashi, H. Kojima, and Z. Furukawa, "Coverage based testing for concurrent software," in Proceedings of the 28th International Conference on Distributed Computing Systems, Beijing, 2008, pp. 533-538.

[71] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka, "Testing concurrent programs: A formal evaluation of coverage criteria," in Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering, Herzliya, 1996, pp. 119-126.

[72] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in Proceedings of the 6th Joint Meeting on European software engineering conference (ESEC) and the ACM SIGSOFT symposium on the foundations of software engineering (FSE-13), Dubrovnik, Croatia, 2007, pp. 533-536.

[73] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simao, and A. C. Hausen, "Structural testing criteria for message-passing parallel programs," *Concurrency and Computation: Practice and Experience,* vol. 20, no. 16, pp. 1893–1916, 2008.

[74] P. S. L. Souza, S. R. S. Souza, and E. Zaluska, “Structural testing for message-passing concurrent programs: an extended test model,” *Concurrency and Computation: Practice and Experience,* vol. 26, no. 1, pp. 21–50, 2014.

[75] M. Popovic, I. Kupresanin, and I. Basicevic, “Generic method for statistical testing of parallel programs based on task trees,” *Scientific Research and Essays,* vol. 7, no. 11, pp. 1244-1255, 2012.

[76] H. Kojima, J. Takahashi, T. Ohta, and Y. Kakuda, “A model for concurrent states and its coverage criteria,” in Proceedings of the International Symposium on Autonomous Decentralized Systems (ISADS), Athens, 2009, pp. 1-6.

[77] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, “Maple: a coverage-driven testing tool for multithreaded programs,” in Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA), Tucson, Arizona, USA, 2012, pp. 485-502.

[78] B. P. Miller, and J. D. Choi, “A mechanism for efficient debugging of parallel programs,” *ACM SIGPLAN Notices,* vol. 23, no. 7, pp. 135-144, 1988.

[79] K. L. McMillan, and D. K. Probst, “A technique of state space search based on unfolding,” *Formal Methods in System Design,* vol. 6, no. 1, pp. 45-65, 1995.

[80] T. Katayama, E. Itoh, Z. Furukawa, and K. Ushijima, “Test case generation for Concurrent programs with the testing criteria using interaction sequences,” in Proceedings of the 6th Asia Pacific software Engineering Conference (APSEC), Takamatsu, 1999, pp. 590-597.

[81] S. K. D. Kamal, and J. M. Francioni, “Nondeterminancy: Testing and debugging in message passing parallel programs.,” *ACM SIGPLAN Notices,* vol. 28, no. 12, pp. 118-128, 1993.

[82] B. Lei, L. Wang, and X. Li, “UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race and Inconsistency,” in Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, Lillehammerpp, 2008, pp. 200-209.

[83] X. Bao, N. Zhang, and Z. Ding, “Test case generation of concurrent programs based on event graphs,” in Proceedings of the Fifth International Joint Conference on INC,IMS and IDC, Seoul, Korea, 2009, pp. 143-149.

[84] A. Ali, A. Nadeem, M. Z. Z. Iqbal, and M. Usman, “Regression Testing based on UML Design Models,” in Proceedings of the 13th Pacific Rim International

Symposium on Dependable Computing (PRDC), Melbourne, Qld., 2007, pp. 85-88.

[85] J. Chen, and S. MacDonald, “Towards a better collaboration of static and dynamic analyses for testing concurrent programs,” in Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging (PADTAD), Seattle, Washington, 2008, pp. 1-9.

[86] P. Wu, and H. Lin, “Model-Based Testing of Concurrent Programs with Predicate Sequencing Constraints,” in Proceedings of the Fifth International Conference on Quality Software (QSIC), Melbourne, Australia, 2005, pp. 3-10.

[87] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, R. Wallentine, and T. Wallentine, “Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs ” *Lecture Notes in Computer Science,* vol. 3920, pp. 73-89, 2006.

[88] A. Luangsodsai, and C. Fox, “Concurrent Statechart Slicing,” in Proceedings of the 2nd Computer Science and Electronic Engineering Conference (CEEC), Colchester, 2010, pp. 1-7.

[89] Z. Chen, B. Xu, K. Liu, H. Yang, and J. Zhang, “An approach to analyzing dependency of concurrent programs,” in Proceedings of the 1st Asia-Pacific Conference on Quality Software (APAQS), Hong Kong, 2000, pp. 39-43.

[90] S. K. Kim, L. Wildman, and R. Duke, “A UML Approach to the Generation of Test Sequences for Java-based Concurrent Systems,” in Proceedings of the Australian Software Engineering Conference (ASWEC), Brisbane, Australia, 2005, pp. 100-109.

[91] J. P. Kitajima, and B. Plateau, “ANDES: a tool for evaluating parallel systems,” in VII Simpósio Brasileiro de Arquitetura de Computadores-Processamento de Alto Desempenho (SBAC-PAD'95), Canela, Brazil, 1995, pp. 367-381.

[92] D. L. Bruening, and J. Chapin, “Systematic testing of multithreaded programs,” Department of Electrical engineering and Computer Science, Massachusetts Institute of Technology, 1999.

[93] S. K. Swain, D. P. Mohapatra, and R. Mall, “Test Case Generation Based on Use case and Sequence Diagram,” *International Journal of Software Engineering,* vol. 3, no. 2, pp. 21-52, 2010.

[94] H. P. Leon, S. Haar, and D. Longuet, "Model Based Testing for Concurrent Systems with Labeled Event Structures," *HAL open archival, hal-00796006, Version 1*, 2013.

[95] V. Garousi, “UML Model-driven Detection of Performance Bottlenecks in Concurrent Real-Time Software,” in International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), Ottawa, 2010, pp. 317-324.

[96] R. Duke, L. Wildman, and B. Long, “Modelling java concurrency with object-Z,” in Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM), Brisbane, Queensland, Australia, 2003, pp. 173-181.

[97] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, and L. K. Dillon, “Debugging Concurrent Software: A Study Using Multithreaded Sequence Diagrams,” in Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Leganes, 2010, pp. 33-40.

[98] Y. Chen, “Generation of Test Cases for Concurrent Software Systems Based on Data-Flow-Oriented Specifications,” in Proceedings of the First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering (CNSI), Jeju Island, 2011, pp. 170-177.

[99] C. A. Sun, “A Transformation-based Approach to Generating Scenario-oriented Test Cases from UML Activity Diagrams for Concurrent Applications,” in Proceedings of the 32nd Annual IEEE International conference on Computer Software and Applications, Turku, 2008, pp. 160-167.

[100] K. Mehner, and A. Wagner, “Visualizing the Synchronization of Java-Thread with UML,” in Proceedings of the IEEE International Symposium on Visual Languages, Seattle, WA, 2000, pp. 199-206.

[101] K. Mehner, “JaVis: A UML-Based Visualization and debugging environment for concurrent java programs ” *Lecture Notes in Computer Science,* vol. 2269, pp. 163-175, 2002.

[102] P. Mehlitz, N. Rungta, and W. Visser, “A hands-on Java PathFinder tutorial,” in Proceedings of the International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 2013, pp. 1493-1495.

[103] K. Havelund, and T. Pressburger, “Model checking JAVA programs using JAVAPathFinder,” *International Journal on Software Tools for Technology Transfer,* vol. 2, no. 4, pp. 366-381, 2000.

[104] S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: A tool for model checking MPI programs," in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP), Salt Lake City, UT, USA, 2008, pp. 285-286.

[105] M. Shousha, L. C. Briand, and Y. Labiche, "A UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems," *IEEE Transactions on Software Engineering,* vol. 38, no. 2, pp. 354-374, 2012.

[106] A. Sen, and M. S. Abadir, "Coverage metrics for verification of concurrent SystemC design using mutation testing," in Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT), Anaheim, FL, 2010, pp. 75-81.

[107] B. Long, R. Duke, D. Goldson, P. Strooper, and L. Wildman, "Mutation based exploration of a method for verifying concurrent java components," in Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, New Mexico, 2004, pp. 265.

[108] B. K. Aichernig, and C. C. Delgado, "From faults via test purposes to test cases: On the fault-based testing of concurrent systems " *Lecture Notes in Computer Science,* vol. 3922, pp. 324-338, 2006.

[109] S. Copty, and S. Ur, "Towards automatic concurrent debugging via minimal program mutant generation with AspectJ," *Electronic Notes in Theoretical Computer Science* vol. 174, no. 9, pp. 151-165, 2007.

[110] M. Delamaro, M. Pezze, and A. M. R. Vincenzi, "Mutant operators for testing concurrent java programs," in Proceedings of the Brazilian Symposium on Software Engineering (SBES), 2001, pp. 272-285.

[111] R. Carver, "Mutation-Based testing of concurrent programs," in Proceedings of the International Test Conference (ITC), Baltimore, MD, 1993, pp. 845-853.

[112] S. Kim, J. A. Clark, and J. A. McDermid, *The rigrous generation of Java mutation operators using HAZOP, Technical Report*, University of York, Heslington, York, 1999.

[113] S. Ghosh, "Towards measurement of testability of concurrent object oriented programs using fault insertion: A preliminary investigation," in Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), Montreal, Canada, 2002, pp. 17-25.

[114] J. S. Bradbury, J. R. Cordy, and J. Dingel, “Mutation operators for concurrent java (J2SE 5.0),” in Proceedings of the Second Workshop on Mutation Analysis, Raleigh, NC, 2006, pp. 11.

[115] J. S. Bradbury, J. R. Cordy, and J. Dingel, “Comparative assessment of testing and model checking using program mutation,” in Proceedings of the Testing:Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAIC PART), Windsor, 2007, pp. 210-222.

[116] M. Gligoric, V. Jagannath, and D. Marinov, “MuTMuT: Efficient exploration for mutation testing of multithreaded code,” in Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST), Paris, 2010, pp. 55-64.

[117] L. Wu, and G. Kaiser, *Empirical study of concurrency mutation operators for java, Technical Report*, Columbia University Computer Science, 2010.

[118] R. A. Silva, S. R. S. Souza, and P. S. L. Souza, “Mutation operators for concurrent programs in MPI,” in Proceedings of the 13th Latin American Test Workshop (LATW), Quito, Ecuador, 2012, pp. 1-6.

[119] M. Musuvathi, S. Qadeer, and T. Ball, *CHESS: A systematic testing tool for concurrent software,* Technical Report, MSR-TR-2007-149, Microsoft Research, Microsoft Corporation, Redmond, WA 98052, 2007.

[120] M. Gligoric, V. Jagannath, Q. Luo, and D. Marinov, “Efficient mutation testing of multithreaded code,” *Software Testing, Verification and Reliability,* vol. 23, no. 5, pp. 375–403, 2013.

[121] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, “Selective mutation testing for concurrent code,” in Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA), Lugano, Switzerland, 2013, pp. 224-234.

[122] J. Cheng, “Slicing Concurrent Program - A graph theoretical approach ” *Lecture Notes in Computer Science,* vol. 749, pp. 223-240, 1993.

[123] J. Zaho, J. Cheng, and K. Ushijima, “Static slicing of concurrent object-oriented programs,” in Proceedings of the 20th International Computer Software and Applications Conference (COMPSAC), Seoul, 1996, pp. 312-320.

[124] J. Zaho, “Multithreaded dependence Graphs for Concurrent Java Programs,” in Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE), Los Angeles, CA, 1999, pp. 13-23.

[125] J. Zhao, “Slicing concurrent Java programs,” in Proceedings of the 7th International Workshop on Program Comprehension (IWPC), Pittsburgh, Pennsylvania, USA, 1999, pp. 126-133.

[126] Z. Guangquan, and R. Mei, “An approach of concurrent object-oriented program slicing based on LTL property,” in Proceedings of the International Conference on Computer Science and Software Engineering (CSSE) (Vol. 2), Wuhan, Hubei, 2008, pp. 650-653.

[127] D. Giffhorn, and C. Hammer, “An Evaluation of Slicing Algorithm for Concurrent Programs,” in Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, Paris, 2007, pp. 17-26.

[128] D. Giffhorn, and C. Hammer, “Precise slicing of concurrent programs - An evaluation of static slicing algorithms for concurrent programs,” *Automated Software Engineering,* vol. 16, no. 2, pp. 197-234, 2009.

[129] D. P. Mohapatra, R. Mall, and R. Kumar, “An efficient technique for dynamic slicing of concurrent java programs,” *Lecture Notes in Computer Science,* vol. 3285, pp. 255-262, 2004.

[130] D. P. Mohapatra, R. Mall, and R. Kumar, “Computing dynamic slices of concurrent object-oriented programs,” *Information and Software Technology,* vol. 47, no. 12, pp. 805-817, 2005.

[131] J. T. Lallchandani, and R. Mall, “Computation of Dynamic Slices for Object-Oriented Concurrent Programs.,” in Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 2005, pp. 8.

[132] D. Goswami, and R. Mall, “Dynamic slicing of concurrent programs ” *Lecture Notes in Computer Science,* vol. 1970, pp. 15-26, 2000.

[133] V. P. Ranganath, and J. Hatcliff, “Pruning interference and ready dependence for slicing concurrent java programs,” *Lecture Notes in Computer Science,* vol. 2985, pp. 39-56, 2004.

[134] D. Goswami, and R. Mall, “Fast slicing of concurrent programs,” *Lecture Notes in Computer Science,* vol. 1745, pp. 38-42, 1999.

[135] D. Goswami, and R. Mall, “A parallel algorithm for static slicing of concurrent programs,” *Concurrency and Computation: Practice and Experience,* vol. 16, no. 8, pp. 751–769, 2004.

[136] G. Jayaraman, V. P. Ranganath, and J. Hatcliff, "Kaveri: Delivering the indus java program slicer to eclipse " *Lecture Notes in Computer Science,* vol. 3442, pp. 269-272, 2005.

[137] V. P. Ranganath, and J. Hatcliff, "Slicing Concurrent Java Programs using Indus and Kaveri," *International Journal on Software Tools for Technology Transfer,* vol. 9, no. 5-6, pp. 489-504, 2007.

[138] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng, "A formal study of slicing for multi-threaded programs with JVM concurrency primitives " *Lecture Notes in Computer Science,* vol. 1694, pp. 1-18, 1999.

[139] X. Qi, X. Zhou, X. Xu, and Y. Zhang, "Slicing Concurrent Programs Based on Program Reachability Graphs," in Proceedings of the 10th International Conference on Quality Software (QSIC), Zhangjiajie, 2010, pp. 248-253.

[140] J. Krinke, "Context Sensitive Slicing of Concurrent Programs," in Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, Helsinki, Finland, 2003, pp. 178-187.

[141] J. Krinke, "Evaluating context-sensitive slicing and chopping," in Proceedings of the International Conference on Software Maintenance (ICSM 2002), Montréal, Canada, 2002, pp. 22-31.

[142] J. Krinke, "Advanced Slicing of Sequential and Concurrent Programs," in Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM), Chicago Illinois, USA, 2004, pp. 464-468.

[143] M. G. Nanda, and S. Ramesh, "Slicing Concurrent Programs," in Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA), Portland, Oregon, United States, 2000, pp. 180-190.

[144] J. Zaho, J. Cheng, and K. Ushijima, "Computing executable slices for concurrent logic programs," in Proceedings of the 2nd Asia-Pacific Conference on Quality Software (APAQS), Hong Kong, 2001, pp. 13-22.

[145] J. Cheng, "Uncertainty Problem In Dynamic Slicing Of Concurrent Programs," in Proceedings of the IEEE International Conference on Embedded Software and Systems, Zhejiang, 2009, pp. 241-248.

[146] N. Rungta, and E. Mercer, "Slicing and Dicing bugs in concurrent programs," in Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE) (Vol. 2), Cape Town, 2010, pp. 195-198.

[147] A. Szegedi, and T. Gyimothy, “Dynamic slicing of java bytecode programs,” in Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), Budapest, Hungary, 2005, pp. 35-44.

[148] M. M. Olm, and H. Seidl, “On optimal slicing of parallel programs,” in Proceedings of the thirty-third annual ACM symposium on Theory of computing, Hersonissos, Greece, 2001, pp. 647-656.

[149] S. Tallam, C. Tian, and R. Gupta, “Dynamic slicing of multithreaded programs for race detection,” in Proceedings of the IEEE International Conference on Software Maintenance, Beijing, 2008, pp. 97-106.

[150] P. Rousseau, “A new approach for concurrent program slicing,” *Lecture Notes in Computer Science,* vol. 4229, pp. 228-242, 2006.

[151] M. B. Dwyer, J. C. Corbett, J. Hatcliff, S. Sokolowski, and H. Zheng, *Slicing multi-threaded java programs: A case study, Technical Report*, Kansas State University Computing and Information Sciences, 1999.

[152] J. Krinke, “Static Slcing of Threaded Programs,” in Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE), Montreal, Quebec, Canada, 1998, pp. 35-42.

[153] Z. Chen, and B. Xu, “Slicing Concurrent Java Programs,” *ACM SIGPLAN Notices,* vol. 36, no. 4, pp. 41-47, 2001.

[154] S. N. Weiss, “A formal framework for the study of concurrent program testing,” in Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, Banff, Alta, 1988, pp. 106-113.

[155] J. Tretmans, “Testing concurrent systems: A formal approach,” *Lecture Notes in Computer Science,* vol. 1664, pp. 46-65, 1999.

[156] T. Bultan, R. Gerber, and W. Pugh, “Model-Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations, and experimental Results,” *ACM Transactions on Programming Languages and Systems,* vol. 21, no. 4, pp. 747-789, 1999.

[157] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, “Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs,” in Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA), Portland, Maine, USA, 2006, pp. 157-168.

[158] C. W. Wang, A. Cavarra, and J. Davies, "Formal and Model-Based Testing of concurrent Workflows," in Proceedings of the 11th International Conference on Quality Software (QSIC), Madrid, Spain, 2011, pp. 252-259.

[159] S. Ray, and R. Sumners, "Specification and Verification of concurrent programs through refinements," *Journal of Automated Reasoning,* vol. 51, no. 3, pp. 241-280, 2013.

[160] V. A. Vasenin, and M. A. Krivchikov, "A model of dynamical concurrent program execution," *Programming and Computer Software,* vol. 39, no. 1, pp. 1-9, 2013.

[161] K. Sen, "Effective Random testing of concurrent programs," in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE), Atlanta, Georgia, USA, 2007, pp. 323-332.

[162] K. Sen, "Race directed random testing of concurrent programs," in Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, Tucson, AZ, USA, 2008, pp. 11-21.

[163] Z. Letko, "Sophisticated Testing of Concurrent Programs," in IEEE Computer Society,Proceedings of the 2nd International Symposium on Search Based Software Engineering, Benevento, Italy, 2010, pp. 36-39.

[164] B. Krena, Z. Letko, T. Vojnar, and S. Ur, "A Platform for Search-Based Testing of concurrent Software," in Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), Trento, Italy, 2010, pp. 48-58.

[165] B. Krena, Z. Letko, and T. Vojnar, "Coverage metrics for saturation-based and search-based testing of concurrent software," *Lecture Notes in Computer Science,* vol. 7186, pp. 177-192, 2012.

[166] A. T. Endo, A. S. Simao, S. R. S. Souza, and P. S. L. Souza, "Web services composition testing-a strategy based on structural testing of parallel programs," in Proceedings of the Testing: Academic & Industrial Conference Practice and Research Techniques, Windsor, 2008, pp. 3-12.

[167] M. Shousha, L. Briand, and Y. Labiche, "A UML/SPT model analysis methodology for concurrent systems based on genetic algorithms " *Lecture Notes in Computer Science,* vol. 5301, pp. 475-489, 2008.

[168] T. Katayama, Z. Furukawa, and K. Ushijima, "A method for structural testing of Ada concurrent programs using the event interactions graph," in Proceedings of

the Asia-Pacific Software Engineering Conference (APSEC), Seoul, 1996, pp. 355-364.

[169] V. Kahlon, S. Sankarranarayanan, and A. Gupta, "Static analysis for concurrent programs with applications to data race detection," *International Journal on Software Tools for Technology Transfer,* vol. 15, no. 4, pp. 321-336, 2013.

[170] S. Gradara, A. Santone, M. L. Villani, and G. Vaglini, "Model Checking Multithreaded Programs by Means of Reduced Models," in Proceedings of the ACM 4th International Workshop on Language Descriptions, Tools and Applications (LDTA), Barcelona, Spain, 2004, pp. 55–74.

[171] D. P. Mohapatra, R. Mall, and R. Kumar, "A Novel Approach for Dynamic Slicing of distributed Object-Oriented Programs," *Lecture Notes in Computer Science,* vol. 3347, pp. 304-309, 2005.

[172] J. Lonnberg, M. B. Ari, and L. Malmi, "Visualising concurrent programs with dynamic dependence graphs," in Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis, Williamsburg, VA, 2011, pp. 1-4.

[173] V. P. Ranganath, and J. Hatcliff, "An Overview of the indus Framework for Analysis and Slicing of Concurrent Java Software," in Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), Philadelphia, Pennsylvania, USA, 2006, pp. 3-7.

[174] W. E. Wong, and Y. Lei, "Reachability graph-based tests sequence generation for concurrent programs," *International Journal of Software Engineering and Knowledge Engineering,* vol. 18, no. 6, pp. 803-822, 2008.

[175] L. Naslavsky, H. Ziv, and D. J. Richardson, "Towards traceability of model-based testing artifacts," in 3rd International workshop on Advances in model-based testing (A-MOST), London, United Kingdom, 2007, pp. 105-114.

[176] S. Weißleder, "Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines," PhD Thesis, Humboldt University of Berlin, 2010.

[177] Y. M. Malik, "Model Based Testing: An Evaluation," Master Thesis, Blekinge Institute of Technology, 2010.

[178] M. A. Mäkinen, "Model Based Approach to Software Testing," Master Thesis, Helsinki University of Technology, 2007.

[179] M. R. Woodward, “Insights into software testing,” *Software Focus,* vol. 2, no. 3, pp. 93-103, 2001.

[180] J. Rilling, and B. Karanth, “A Hybrid Program Slicing Framework,” in 1st IEEE International Workshop on Source Code Analysis and Manipulation, Florence, Italy, 2001, pp. 12-23.

[181] P. C. Attie, “Efficient formal methods for the synthesis of concurrent programs,” *ACM SIGSOFT Software Engineering Notes,* vol. 25, no. 1, pp. 34, 2000.

[182] F. Xhafa, and A. Abraham, *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*: Springer-Verlag Berlin Heidelberg, 2008.

[183] X. S. Yang, *Nature-inspired optimization algorithms*: Elsevier, 2014.

[184] A. E. Hassanien, and E. Emary, *Swarm Intelligence: Principles, Advances, and Applications*: CRC Press, Taylor and Francis Group, LLC, 2016.

[185] E. G. Talbi, *Metaheuristics from Design to Implementation*: John Wiley & Sons, 2009.

[186] C. Blum, and A. Roli, “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM Computing Surveys (CSUR),* vol. 35, no. 3, pp. 268-308, 2003.

[187] Z. Michalewicz, and D. B. Fogel, *How to Solve It: Modern Heuristics*: Springer, 2004.

[188] *Decision Sciences: Theory and Practice*: CRC Press, Taylor & Francis Group, 2017.

[189] M. Shirole, M. Kommuri, and R. Kumar, “Transition sequence exploration of UML activity diagram using evolutionary algorithm,” in Proceedings of 5th India Software Engineering Conference, 2012, pp. 97-100.

[190] A. Nayak, and D. Samanta, “Synthesis of test scenarios using UML activity diagrams,” *Software & Systems Modeling,* vol. 10, no. 1, pp. 63-89, 2011.

[191] D. Kundu, and D. Samanta, “A novel approach to generate test cases from UML activity diagrams,” *Journal of object technology,* vol. 8, no. 3, pp. 65-83, 2009.

[192] P. R. Srivastava, K. Baby, and G. Raghurama, “An Approach of Optimal Path Generation using Ant Colony Optimization,” in IEEE Region 10 Conference, Singapore, 2009, pp. 1-6.

[193] H. Li, and C. P. Lam, “Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams,” *Testing of Communicating Systems, Lecture Notes in Computer Science, Springer Berlin Heidelberg,* vol. 3502 pp. 69-80, 2005.

[194] C. P. Lam, "Computational Intelligence for Functional Testing," *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*, F. Meziane, ed.: IGI Global, 2010.

[195] U. Farooq, C. P. Lam, and H. Li, “Towards Automated Test Sequence Generation,” in Proceedings of 19th Australian Conference on Software Engineering, 2008, pp. 441 – 450.

[196] A. Mishra, and D. P. Mohapatra, “Generation and Prioritization of test sequences using UML activity diagram,” B.Tech. Dissertation, National Institute of Technology, Rourkela, India, 2014.

[197] F. Sayyari, and S. Emadi, “Automated generation of software testing path based on ant colony,” in 2nd International Congress on Technology, Communication and Knowledge (ICTCK), Islamic Azad University, Mashhad, Iran, 2015, pp. 435-440.

[198] A. Adamatzky, *Advances in Physarum machines: Sensing and computing with slime mould*: Springer, 2016.

[199] J. Jones, “Mechanisms inducing parallel computation in a model of Physarum polycephalum transport networks,” *Parallel Processing Letters,* vol. 25, no. 1, 2015.

[200] Y.-P. Gunji, T. Shirakawa, T. Niizato, M. Yamachiyo, and I. Tani, “An adaptive and robust biological network based on the vacant-particle transportation model,” *Journal of theoretical biology,* vol. 272, no. 1, pp. 187-200, 2011.

[201] Y. Singh, A. Kaur, and B. Suri, “Test case prioritization using ant colony optimization,” *ACM SIGSOFT Software Engineering Notes,* vol. 35, no. 4, 2010.

[202] H. Wang, Z. Shi, and S. Li, “Multicast routing for delay variation bound using a modified ant colony algorithm,” *Journal of Network and Computer Applications,* vol. 32, no. 1 pp. 258-272, 2009.

[203] S. Garnier, A. Guérécheau, M. Combe, V. Fourcassié, and G. Theraulaz, “Path selection and foraging efficiency in Argentine ant transport networks,” *Behavioral Ecology and Sociobiology,* vol. 63, no. 8 pp. 1167-1179, 2009.

[204] C. Zhang, Z. Zhen, D. Wang, and M. Li, “UAV path planning method based on ant colony optimization,” in IEEE Chinese Control and Decision Conference (CCDC), Xuzhou, China, 2010, pp. 3790-3792.

[205] J. Branke, and M. Guntsch, "Solving the probabilistic TSP with ant colony optimization," *Journal of Mathematical Modelling and Algorithms,* vol. 3, no. 4 pp. 403-425, 2004.

[206] B. Xing, and G. W. Jing, *Innovative computational intelligence: a rough guide to 134 clever algorithms*: Springer Cham Heidelberg, 2014.

[207] X. Zhang, Y. Zhang, Z. Zhang, S. Mahadevan, A. Adamatzky, and Y. Deng, "Rapid Physarum Algorithm for shortest path problem," *Applied Soft Computing,* vol. 23, pp. 19-26, 2014.

[208] X. Zhang, Q. Wang, A. Adamatzky, F. T. Chan, S. Mahadevan, and Y. Deng, "A biologically inspired optimization algorithm for solving fuzzy shortest path problems with mixed fuzzy arc lengths," *Journal of optimization theory and applications,* vol. 163, no. 3, pp. 1049-1056, 2014.

[209] X. Zhang, A. Adamatzky, H. Yang, S. Mahadaven, X. S. Yang, Q. Wang, and Y. Deng, "A bio-inspired algorithm for identification of critical components in the transportation networks," *Applied Mathematics and Computation,* vol. 248, pp. 18-27, 2014.

[210] A. Schumann, and A. Adamatzky, "Physarum spatial logic," *New Mathematics and Natural Computation,* vol. 7, no. 3, pp. 483-498, 2011.

[211] A. Schumann, and K. Pancerz, "Towards an object-oriented programming language for Physarum Polycephalum computing: a petri net model approach," *Fundamenta Informaticae,* vol. 133, no. 2-3, pp. 271-285, 2014.

[212] K. Pancerz, and A. Schumann, "Some issues on an object-oriented programming language for Physarum machines," *In Applications of Computational Intelligence in Biomedical Technology, Springer International Publishing*, pp. 185-199, 2016.

[213] A. Tero, R. Kobayashi, and T. Nakagaki, "A mathematical model for adaptive transport network in path finding by true slime mold," *Journal of Theoretical Biology,* vol. 244, pp. 553–564, 2007.

[214] X. Zhang, S. Huang, Y. Hu, Y. Zhang, S. Mahadevan, and Y. Deng, "Solving 0-1 knapsack problems based on amoeboid organism algorithm," *Applied Mathematics and Computation,* vol. 219, no. 19, pp. 9959-9970, 2013.

[215] Y. Liu, C. Gao, Z. Zhang, Y. Wu, M. Liang, L. Tao, and Y. Lu, "A new multi-agent system to simulate the foraging behaviors of Physarum," *Natural Computing*, pp. 1-15, 2015.

[216] X. Zhang, Q. Wang, A. Adamatzky, F. T. S. Chan, S. Mahadevan, and Y. Deng, "An improved physarum Polycephalum algorithm for the shortest path problem," *The Scientific World Journal*, pp. 9 pages, 2014.

[217] L. Masi, and M. Vasile, *Multidirectional Physarum Solver: an Innovative Bio-inspired Algorithm for Optimal Discrete Decision Making,* Report 01012013, University of Strathclyde, Glasgow, United Kingdom, 2013.

[218] X. Zhang, Q. Wang, F. T. Chan, S. Mahadevan, and Y. Deng, "A Physarum Polycephalum optimization algorithm for the bi-objective shortest path problem," *International Journal of Unconventional Computing,* vol. 10 no. 1/2, pp. 143-162, 2014.

[219] X. Zhang, A. Adamatzky, F. T. Chan, Y. Deng, H. Yang, X. S. Yang, M. A. I. Tsompanas, G. C. Sirakoulis, and S. Mahadevan, "A biologically inspired network design model," *Scientific reports 5*, 2015.

[220] A. Schumann, K. Pancerz, A. Adamatzky, and M. Grube, "Bio-inspired game theory: the case of physarum Polycephalum," in Proceedings of the 8th International Conference on Bioinspired Information and Communications Technologies, Boston, Massachusetts, 2014, pp. 9-16.

[221] K. Pancerz, and A. Schumann, "Rough set models of Physarum machines," *International Journal of General Systems,* vol. 44, no. 3, pp. 314-325, 2015.

[222] K. Pancerz, and A. Schumann, "Rough Set Description of Strategy Games on Physarum Machines," in Advances in Unconventional Computing: Volume 2: Prototypes, Models and Algorithms, 2017, pp. 615-636.

[223] A. Tero, R. Kobayashi, and T. Nakagaki, "Physarum solver: a biologically inspired method of road-network navigation," *Physica A: Statistical Mechanics and its Applications,* vol. 363, no. 1, pp. 115-119, 2006.

[224] M. Črepinšek, S. H. Liu, and M. Mernik, "Replication and comparison of computational experiments in applied evolutionary computing: common pitfalls and guidelines to avoid them," *Applied Soft Computing,* vol. 19, pp. 161-170, 2014.

[225] A. Arcuri, and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," *Lecture Notes in Computer Science, Springer Berlin Heidelberg,* vol. 6956, pp. 33-47, 2011.

[226] R. Karimpour, and G. Ruhe, “Evolutionary robust optimization for Software Product Line scoping: An explorative study,” *Computer Languages, Systems & Structures,* vol. 47, pp. 189-210, 2017.

[227] A. Mishra, and P. Kumar Mishra, “A Randomized Scheduling Algorithm for Multiprocessor Environments Using Local Search,” *Parallel Processing Letters,* vol. 26, no. 1, pp. 1650002-1650018, 2016.

[228] M. Dorigo, and T. Stützle, *Ant Colony Optimization*, Cambridge, Massachusetts: The MIT Press, 2004.

[229] M. Mernik, S. H. Liu, D. Karaboga, and M. Črepinšek, “On clarifying misconceptions when comparing variants of the Artificial Bee Colony Algorithm by offering a new implementation,” *Information Sciences,* vol. 291, pp. 115-127, 2015.

[230] Amarjeet, and J. K. Chhabra, “Harmony search based remodularization for object-oriented software systems,” *Computer Languages, Systems & Structures,* vol. 47, pp. 153-169, 2017.

[231] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, “The Quest for Open Source Projects that Use UML: Mining GitHub,” in ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, 2016, pp. 173-183.

[232] "UML Activity Diagram Repository," 10th September, 2016; https://github.com/UMLADRepo/AD-Repository.

[233] J. Derrac, S. García, D. Molina, and F. Herrera, “A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms,” *Swarm and Evolutionary Computation,* vol. 1, no. 1, pp. 3-18, 2011.

[234] H. Garg, “A hybrid PSO-GA algorithm for constrained optimization problems,” *Applied Mathematics and Computation,* vol. 274, pp. 292–305, 2016.

[235] R. Rosengren, “Route Fidelity, Visual Memory and Recruitment Behaviour in Foraging Wood Ants of the Genus Formica:(Hymenoptera, Formicidae),” *Societas pro fauna et flora Fennica*, 1971.

[236] A. Prakasam, and N. Savarimuthu, “Metaheuristic algorithms and probabilistic behaviour: a comprehensive analysis of Ant Colony Optimization and its variants,” *The Artificial Intelligence Review,* vol. 45, no. 1, pp. 97, 2016.

[237] W. J. Gutjahr, “A graph-based ant system and its convergence,” *Future generation computer systems,* vol. 16, no. 8, pp. 873-888, 2000.

[238] J. L. Deneubourg, J. M. Pasteels, and J. C. Verhaeghe, “Probabilistic behaviour in ants: a strategy of errors?,” *Journal of Theoretical Biology,* vol. 105, no. 2, pp. 259-271, 1983.

[239] M. Dorigo, V. Maniezzo, and A. Colorni, *The ant system: An autocatalytic optimizing process,* Technical report: 91-016, 1991.

[240] S. Christodoulou, “Scheduling resource-constrained projects with ant colony optimization artificial agents,” *Journal of Computing in Civil Engineering,* vol. 24, no. 1, pp. 45-55, 2009.

[241] M. D. Toksari, “A hybrid algorithm of ant colony optimization (ACO) and iterated local search (ILS) for estimating electricity domestic consumption: case of Turkey,” *International Journal of Electrical Power & Energy Systems,* vol. 78, pp. 776-782, 2016.

[242] T. Stützle, M. López-Ibánez, P. Pellegrini, M. Maur, M. Antonio Montes de Oca, M. Birattari, and M. Dorigo, *Parameter Adaptation in Ant Colony Optimization*: Springer Berlin Heidelberg, 2012.

[243] E. Sun, C. Wang, and F. Tian, “A Survey on Multi-path Routing Protocols in Wireless Multimedia Sensor Networks,” *Indonesian Journal of Electrical Engineering and Computer Science,* vol. 12, no. 9, pp. 6978-6983, 2014.

[244] K. Sundareswaran, K. Jayant, and T. N. Shanavas, “Inverter harmonic elimination through a colony of continuously exploring ants,” *IEEE Transactions on Industrial Electronics,* vol. 54, no. 5 pp. 2558-2565, 2007.