Concurrent coverage criteria for activity diagrams

Mahesh Shrirole1 | Rajeev Kumar2

1Computer Engineering & Information Technology Department, Veerama Jijbabai Technological Institute, Mumbai, India
2School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi, India

Correspondence
Mahesh Shrirole, Computer Engineering & Information Technology Department, Veerama Jijbabai Technological Institute, Matunga, Mumbai 400 019, India.
Email: mrsrirole@it.vjti.ac.in

Abstract
Existing coverage criteria for Unified Modeling Language (UML)-based system design have used mostly structural constructs and sequential-behavioural coverage criteria. However, concurrent-behavioural coverage criteria are not sufficiently explored. The basic path coverage criteria, in UML-based system design, do not distinguish between non-interleaving and interleaving execution paths in concurrent activity diagrams. Herein, the main contribution is to present a new set of coverage criteria, based on activity-interleaving paths for concurrent activity diagrams. For this, first, we modified the definitions of basic path and basic path coverage criterion, and then we defined the interleaving activity path, interleaving activity path coverage, and total activity path coverage criteria. It is observed that the proposed coverage criteria explore most of the concurrency behaviours to uncover concurrency errors. They help to uncover different types of concurrency errors like synchronization, data-race, and deadlock.

1 | INTRODUCTION

With the advent of multi-core architectures, concurrency plays a significant role in design and development of state-of-art software systems. Software testing used for estimating quality and reliability of software under test, is becoming more challenging. The difficulty of concurrent program testing is intensified due to concurrency features such as non-determinism, synchronization, and inter-process communication [1]. As implementation of concurrent designs leads to possibly exponential concurrent behaviours (execution interleaving) at run-time, a proper selection of coverage criteria is a key factor for efficacy and efficiency of the testing efforts.

Concurrent programs lead to potentially non-deterministic interleavings at runtime. This mainly seems to be caused by programmers inability to move away from a flat imperative programming style that was never meant for implementing concurrent behaviour. Threads, semaphores etc. are operating system-specific additions that were added to programming languages that follow a linear execution path. Programmers may fail to obey locking discipline required for the said new operating system-specific additions, this leads to run-time problems. To ease of understanding, one can model concurrent behaviour at a higher level of abstraction, for example using activity diagrams. At that level, a more natural way of specifying such behaviour, using fork/join constructs, is available. From these models, code is ultimately generated in an imperative programming language, in which concurrency issues can occur due to unexpected interleavings.

Concurrency errors are difficult to uncover by traditional testing methods alone, specifically traditional structural testing. Traditional structural testing focuses on structural element coverage: statement, branch, and basic path, which constitutes sequential behavioural coverage criteria. Since, concurrent software system involves execution interleaving inside concurrency constructs, testing of concurrent systems is challenging to uncover concurrency errors. Recently, researchers are exploring on test scenarios generation based on concurrent activity diagrams to explore concurrent behaviour of the system [2–4].

In traditional code-based testing, source code is arranged first into a control flow graph (CFG). Then, coverage criteria based on CFG constructs, are defined by node-coverage, transition coverage, branch coverage, and path coverage [5]. Some of the coverage metrics are also based on the data flow information through CFG such as def-use, def-predicate-use, and def-all-use [6]. These coverage criteria are strong enough to test the traditional structural testing goals because control flow and data flow in CFG are deterministic.

However, traditional structural coverage criteria are not appropriate for concurrent systems because these criteria do not specify requirements on concurrent structural elements and concurrent behaviour. Coverage criteria can be classified in...
two types: (i) structural element coverage criteria, and (ii) behavioural coverage criteria. Structural element coverage criteria are defined on independent elements of either a code or a model. A behavioural coverage criterion is defined on a sequence of structural elements that define a complete behaviour or a functionality of the system. In a concurrency setting, tasks from different threads are executed simultaneously at runtime thereby generating a sequence/arrangement of intermixed tasks from participating threads called as an interleaving path, which represents a concurrent behaviour. Thus, concurrent systems need concurrent behavioural coverage criteria in addition to structural coverage criteria.

Unified Modeling Language (UML) design models promote early test case generation using different design artefacts. UML-based testing techniques derive test requirements and coverage criteria from UML design diagrams. UML designs represent sequential and concurrent behaviours of the system. For activity diagrams, a set of coverage criteria are defined in literature: activity, transition, basic path coverage [7], and simple path coverage [8]. However, concurrent behaviour coverage criteria have not been discussed in detail, for activity and sequence diagrams, which represent concurrent behaviours of systems [9].

In this work, we have attempted to answer the following research questions:

R1 Are existing coverage criteria sufficient to understand concurrent behaviour by fork-join constructs in an activity diagram?

R2 Can the concurrent coverage criteria be proposed for exploring interleaving within a fork-join construct of an activity diagram?

The main contribution is to propose behavioural coverage criteria for concurrency testing based on concurrent activity diagrams. We have extended code-based concurrency coverage criteria to UML activity diagrams. We have presented interleaving activity path coverage (IAPC) and total activity path coverage (TAPC) criteria. The interleaving coverage criteria are inspired from thread-pair-interleaving as in Lu et al. [10]. In addition, we have redefined the basic path to distinguish between non-interleaving and interleaving execution paths for concurrent activity diagrams.

Herein, Section 2 presents a brief overview of the related work. Section 3 includes a motivating example having interleaved activities. Section 4 discusses existing coverage criteria. Section 5 presents the proposed concurrency coverage criteria for concurrent activity diagrams. Section 6 discusses the results. Finally, Section 7 provides the conclusion.

2 | RELATED WORK

There are several heuristic means to measure the quality of test suites like fault detection, mutation analysis, or coverage criteria. Among these, coverage criteria are commonly used and popular heuristic means to measure the quality of the test suite [11]. For UML behavioural models, several researchers have proposed coverage criteria as per their interpretations and requirements.

For activity diagrams, a set of coverage criteria are defined in the literature based on activity, transition, basic path [7], and simple path [8]. These defined coverage criteria include structural element coverage, sequential (non-concurrent) behaviour and concurrent behaviour coverage criteria. However, concurrent behaviour coverage criteria are not explored in detail to address concurrency errors.

In code-based testing, execution interleavings are explored to define concurrency criteria. Code-based concurrency behavioural criteria are studied by [10,12,13]. For UML-based testing, coverage criteria are studied in previous works [7,8,14–17,18]. But, they do not explore concurrency constructs and concurrent behaviours in detail.

Linzhang et al. [7] have presented basic path coverage for activity diagrams. In their approach, an activity diagram is traversed from the initial activity node to the final activity node by a depth-first search (DFS) with all activities and transitions visited where loops must be visited at most ones. These paths represent a potential single sequence of computational logic (functional requirement). These paths are used for structural testing to satisfy the sequential (non-concurrent) behaviour of software or models. This coverage criterion is not related to concurrency testing, and it does not process or includes any concurrency construct.

Rountev et al. [18] have presented control-flow-based coverage criteria for UML sequence diagrams. Their criteria are generalizations of traditional control-flow criteria, such as branch coverage and path coverage. They are defined in terms of the sequence of messages exchanged among the set of collaborating objects. An inter-procedural restricted control-flow graph (IRCFG) is used to define family of test coverage criteria. They have presented All-RCFG-Paths and All-RCFG-Branes coverage criteria for sequence diagrams.

Mingsong et al. [8] have generalized basic path coverage for concurrency notion in activity diagrams. If each activity in the path occurs only once, then such a path is a basic path of the activity diagram. The diagram composed by all the basic paths of an activity diagram is called basic activity diagram [8]. They have proposed to restrict to one representative path among all concurrent execution interleaving paths having the same partial order as a simple path from the basic activity diagram.

Kundu and Samanta [17] have presented basic path and activity path coverage criterion for activity diagrams. They have presented basic path coverage criterion, where they suggested checking loop conditions for true and false paths. For concurrent activity diagrams, they have presented activity path coverage criterion, where they have suggested selecting of concurrent path generated from DFS-BFS algorithm.

Ferreira et al. [15] have presented a state machine model coverage analysis tool. Their tool supports coverage analysis of Modified Condition/Decision Coverage (MC/DC) and boundary-value criteria of the guards of the state machine. It analyzes the coverage of the model that is achieved by a given test suite. For better visual understanding of coverage, it is represented by colouring the elements in the model. Kosmatov et al. [16] have presented data-oriented model-based coverage criteria, by formalizing boundary testing heuristics. A test
selection algorithm for a given boundary coverage criterion is presented. Their algorithm generates test sets by minimizing and maximizing cost functions. These criteria are used to measure the coverage of test cases generated from formal specification models like B, Z and UML/OCL.

For sequence diagrams, a set of coverage criteria are defined, namely message, message sequence path coverage [14], and all message path coverage [19].

Although, existing UML-based coverage criteria are sufficient for non-concurrent systems, there is a need to define the concurrent behavioural coverage criteria for UML design (both activity and sequence diagrams) to address concurrency errors. These criteria enables designing test suite early in the software development to uncover concurrency errors.

3 | A MOTIVATING EXAMPLE

A behaviour represented by an activity diagram is a path from the start node to the end node. There are typically many possible paths between the start node and the end node of activity diagrams, and each path representing a different behaviour of the system at run-time. For testing concurrency errors, one may need to test as large as $n!$ execution interleavings to detect a concurrency error, where $n$ is the degree of parallelism [20]. A degree of parallelism is represented by the number of activities inside a fork-join control structure for concurrent activity diagrams. Non-interleaving execution (sequential execution), where tasks from a single thread are executed from the start node to the end node, always follow a fixed path for a given input. While interleaving execution (parallel execution), where tasks from two or more threads are intermixed randomly from the thread start to the thread end (fork-join) thereby generating different paths for a given input.

We consider an example of a bank account transfer transaction through an automated teller machine (ATM), as shown in Figure 1. This example is selected to illustrate sequential (non-interleaving) and interleaving execution behaviours. The ATM system communicates with the bank server for different transaction types: deposit, withdrawal, amount transfer, and balance inquiry. When a bank customer arrives at an ATM station, he/she inserts a plastic card encoded with the personal account number containing the customer’s name and account information. Then the customer enters his or her personal identification number. The customer selects the desired transaction from the list of options to complete his/her transaction, further in the sequence; the customer’s request is processed by the system. For example, if the customer requests for a money-transfer transaction and enters the destination account number and amount then the system checks the balance in the customer’s account. If the account balance is sufficient, money is transferred to the other account at bank server. The transaction information and new balance are printed on the receipt and the customer’s ATM card is returned. A scenario captured in the figure represents concurrent transaction processing from different ATMs transferring amount between two accounts.

Test case generation from UML designs is governed by the intermediate representation (IR) to which diagrams are transformed. In this approach, we first translate an activity diagram into an IR called activity CFG, which is CFG-like representation suitable for graph traversal algorithms, as shown in Figure 2.

The activity diagram shown in Figure 1 represents four different behaviours: (a) invalid user, (b) valid user with exit action, (c) valid user with insufficient balance, and (d) valid user with sufficient balance. Among these behaviours, fourth behaviour, i.e. a valid user with sufficient balance has a concurrency construct, which leads to six different interleaving paths. The scenarios represented for the said behaviours are discussed based on intermediate graph representation.

Therefore, there are total nine test scenarios representing sequential and concurrent behaviours. All the nine behaviours are listed as follows:

1. $B_1 = S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow M_1 \rightarrow a_1 \rightarrow E.$
2. $B_2 = S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow a_3 \rightarrow D_2 \rightarrow M_1 \rightarrow a_3 \rightarrow E.$
3. $B_3 = S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow a_3 \rightarrow D_2 \rightarrow a_4 \rightarrow D_3 \rightarrow a_5 \rightarrow M_1 \rightarrow a_5 \rightarrow E.$
4. $B_4 = S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow a_3 \rightarrow D_2 \rightarrow a_3 \rightarrow D_3 \rightarrow F \rightarrow a_5 \rightarrow F \rightarrow a_3 \rightarrow D_1 \rightarrow a_3 \rightarrow a_5 \rightarrow E.$

**FIGURE 1** An activity diagram for ATM Transfer Transactions
5. \( B_5^2 = S \xrightarrow{t_1} a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} D_1 \xrightarrow{t_4} a_3 \xrightarrow{t_5} D_2 \xrightarrow{t_6} a_4 \xrightarrow{t_7} D_3 \xrightarrow{t_8} a_5 \xrightarrow{t_9} F \xrightarrow{t_{10}} a_6 \xrightarrow{t_{11}} F \xrightarrow{t_{12}} a_7 \xrightarrow{t_{13}} F \xrightarrow{t_{14}} a_8 \xrightarrow{t_{15}} F \xrightarrow{t_{16}} a_9 \xrightarrow{t_{17}} F \xrightarrow{t_{18}} a_{10} \xrightarrow{t_{19}} F \xrightarrow{t_{20}} a_{11} \xrightarrow{t_{21}} F \xrightarrow{t_{22}} a_{12} \xrightarrow{t_{23}} E \), where a superscript indicates an interleaving number and a subscript indicates a behaviour number.

For non-concurrent programs, there is no notion of a context/thread switch among the parts of the same program. Concurrent programs have more than one thread, control flow from one thread to another is known as context/thread switch. In a test scenario of a concurrent behaviour, a context switch is represented by a keyword CS below the arrow, like \( a_7 \xrightarrow{t_{14}} a_8 C_S \). The fourth behaviour has six interleavings: \( B_1^1, B_1^2, B_1^3, B_1^4, B_1^5, \) and \( B_1^6 \), where a superscript indicates an interleaving number and a subscript indicates a behaviour number.

4 | EXISTING COVERAGE CRITERIA

Next, we consider the exiting coverage criteria for activity diagrams and consider the above example to appreciate the coverage. Linzhang et al. [7] defined basic path coverage for activity diagrams. Their basic path definition allows all distinct paths generated by the Depth First Search (DFS) method from the initial activity to the final activity. A set of basic paths must cover all transitions and action states. It has only restriction to the loop nodes, which may generate multiple paths. The restriction is to execute loops at the most once thereby generating only one path for the loop nodes. Kundu and Samanta [17] have illustrated the need of testing both the looping and non-looping paths for loop nodes.

Basic path coverage criterion defined in Linzhang et al.’s approach is as follows:

**Definition 1** Basic Path Coverage Criterion: Let BP be the basic path set of an activity diagram, tc be the set of test cases, for any \( p_i \in BP \), there must be at least one test case \( t \in tc \) such that when the software is executed using ‘t’, the basic path ‘\( p_i \)’ of the activity diagram is executed [7].

A basic path represents an independent path from the start node to the end node of either a graph or a graph representation of an activity diagram. An independent path represents a potential single sequence of computational logic (functional requirements). A set of all such independent paths are used for structural testing to satisfy the sequential (non-concurrent) behaviour of software or models. In their definition of the basic path, processing of fork-join construct is not addressed. Hence, basic path coverage criterion is not related to concurrency testing, and it does not process or include any concurrency artefact.

For the motivating example’s activity CFG in Figure 2, DFS method generates five paths from the initial activity to the end activity as follows.

1. \( p_1 = S \xrightarrow{t_1} a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} D_1 \xrightarrow{t_4} a_3 \xrightarrow{t_5} D_2 \xrightarrow{t_6} a_4 \xrightarrow{t_7} M_1 \xrightarrow{t_8} a_{11} \xrightarrow{t_{15}} F \xrightarrow{t_{16}} a_{12} \xrightarrow{t_{17}} F \xrightarrow{t_{18}} a_{13} \xrightarrow{t_{19}} M_1 \xrightarrow{t_{20}} a_{11} \xrightarrow{t_{21}} E \).
2. \( p_2 = S \xrightarrow{t_1} a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} D_1 \xrightarrow{t_4} a_3 \xrightarrow{t_5} D_2 \xrightarrow{t_6} a_4 \xrightarrow{t_7} M_1 \xrightarrow{t_8} a_{11} \xrightarrow{t_{15}} F \xrightarrow{t_{16}} a_{12} \xrightarrow{t_{17}} F \xrightarrow{t_{18}} a_{13} \xrightarrow{t_{19}} M_1 \xrightarrow{t_{20}} a_{11} \xrightarrow{t_{21}} E \).
3. \( p_3 = S \xrightarrow{t_1} a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} D_1 \xrightarrow{t_4} a_3 \xrightarrow{t_5} D_2 \xrightarrow{t_6} a_4 \xrightarrow{t_7} D_1 \xrightarrow{t_8} a_5 \xrightarrow{t_{10}} M_1 \xrightarrow{t_{11}} a_{11} \xrightarrow{t_{15}} F \xrightarrow{t_{16}} a_{12} \xrightarrow{t_{17}} F \xrightarrow{t_{18}} a_{13} \xrightarrow{t_{19}} M_1 \xrightarrow{t_{20}} a_{11} \xrightarrow{t_{21}} E \).
4. \( p_4 = S \xrightarrow{t_1} a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} D_1 \xrightarrow{t_4} a_3 \xrightarrow{t_5} D_2 \xrightarrow{t_6} a_4 \xrightarrow{t_7} D_1 \xrightarrow{t_8} a_5 \xrightarrow{t_{10}} M_1 \xrightarrow{t_{11}} a_{11} \xrightarrow{t_{15}} F \xrightarrow{t_{16}} a_{12} \xrightarrow{t_{17}} F \xrightarrow{t_{18}} a_{13} \xrightarrow{t_{19}} M_1 \xrightarrow{t_{20}} a_{11} \xrightarrow{t_{21}} E \).
5. \( p_5 = S \xrightarrow{t_1} a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} D_1 \xrightarrow{t_4} a_3 \xrightarrow{t_5} D_2 \xrightarrow{t_6} a_4 \xrightarrow{t_7} D_3 \xrightarrow{t_8} F \xrightarrow{t_{15}} a_5 \xrightarrow{t_{10}} F \xrightarrow{t_{16}} a_6 \xrightarrow{t_{17}} F \xrightarrow{t_{18}} a_7 \xrightarrow{t_{19}} F \xrightarrow{t_{20}} a_8 \xrightarrow{t_{21}} F \xrightarrow{t_{22}} E \).
Paths $p_1$, $p_2$, and $p_3$ represent three sequential (non-concurrent) behaviours $B_1$, $B_2$, and $B_3$ respectively. Paths $p_4$ and $p_5$ represent behaviour $B_4$, but behaviour $B_4$ is concurrent and paths $p_4$ and $p_5$ do not represent complete concurrent execution paths. The concurrent behaviour $B_4$, which has six execution interleaveings, is not distinguished from sequential behaviour in the above definition of basic path. Basic path definition does not give clear idea whether interleaving paths of concurrent behaviours are to be considered or not. Thus basic path coverage criterion does not give clear estimation of the coverage of paths in case of concurrent activity diagrams.

To overcome the issue of the concurrent behaviours, Mingsong et al. [8] have generalized basic path coverage using the partial order relation between activities inside the fork-join construct of activity diagrams. They have proposed simple path definition based on the basic activity diagram, which is composed by all the basic paths of an activity diagram. Mingsong et al. have used improved DFS algorithm to take care of concurrency constructs thereby generating complete concurrent paths. Thus, all the paths generated by the improved DFS algorithm considers sequential behaviours and concurrent behaviours. The definition of simple path presented in their approach uses all sequential and concurrent paths generated from an activity diagram. From all these paths they select distinct paths that have a different set of activities and their partial order. In other words, they select one representative path from a set of paths that have the same set of activities and the same partial order relation. Thus, simple path includes all basic paths as defined in Linzhang et al.’s approach for sequential behaviour and one representative path from all concurrent paths that have the same set of activities and the same partial order relation for concurrent behaviour.

Consider the following example to understand selection of one representative path among all concurrent execution interleaving paths having the same partial order as a simple path. Suppose threads $T_1 = \{a_1, a_2\}$ and $T_2 = \{b_1, b_2\}$, where $a_1$, $a_2$, $b_1$, and $b_2$ represent activities in the respective threads inside fork-join concurrency construct then paths $s_1 = a_1 \rightarrow a_2 \rightarrow b_1 \rightarrow b_2$, $s_2 = a_1 \rightarrow b_1 \rightarrow a_2 \rightarrow b_2$, $s_3 = a_1 \rightarrow b_1 \rightarrow b_2 \rightarrow a_2$, $s_4 = b_1 \rightarrow a_1 \rightarrow b_2 \rightarrow a_2$, $s_5 = b_1 \rightarrow a_1 \rightarrow a_2 \rightarrow b_2$, and $s_6 = b_1 \rightarrow a_1 \rightarrow b_2 \rightarrow a_2$ are all execution interleaveings. All these paths have a set of activities $\{a_1, a_2, b_1, b_2\}$ from threads $T_1$ and $T_2$, and they have same partial order relations: $a_1 \leq a_2$ and $b_1 \leq b_2$, where symbol “$\leq$” represents a partial order relation. Hence, all paths $\{s_1, s_2, s_3, s_4, s_5, s_6\}$ have the same partial order. Any selected single representative path, say $s_1$, is a simple path.

For the motivating example in Figure 1, following four simple paths of which three paths are $\{p_1, p_2, p_3\}$ that cover all behaviours $B_1, B_2, B_3$, and the fourth path is $p_4'$, which covers concurrent behaviour $B_4$.

1. $p_4' = S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow a_3 \rightarrow D_2 \rightarrow a_4 \rightarrow D_3 \rightarrow F \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \rightarrow a_9 \rightarrow a_{10} \rightarrow M_1 \rightarrow a_{11} \rightarrow E$. 

   The path $p_4'$ is a representative simple path for all six interleaving paths. Kundu and Samanta [17] have presented activity path coverage criterion based on simple path coverage criterion where they suggest selecting a path generated by BFS algorithm as a representative path rather than selecting any arbitrary path. Thus selection of a specific representative path among the set of simple path is resolved by considering activity path coverage criterion. BFS algorithm generates the number of paths in proportion to the factorial of number of threads. For example, if fork-join construct has two threads then number of paths generated by BFS algorithm would be $2! = 2$.

For the motivating example in Figure 1, activity path coverage criterion selects four paths of which three paths are $\{p_1, p_2, p_3\}$ and one representative path from following two paths.

1. $p_4 = S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow a_3 \rightarrow D_2 \rightarrow a_4 \rightarrow D_3 \rightarrow F \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \rightarrow a_9 \rightarrow a_{10} \rightarrow M_1 \rightarrow a_{11} \rightarrow E$.
2. $p_5 = S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow a_3 \rightarrow D_2 \rightarrow a_4 \rightarrow D_3 \rightarrow F \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \rightarrow a_9 \rightarrow a_{10} \rightarrow M_1 \rightarrow a_{11} \rightarrow E$.

Concurrency errors, such as deadlock, race condition, and atomicity violation, are not present in traditional sequential programming. A deadlock occurs when a chain of processes/threads are involved in a cycle in which each process is waiting for resources/locks that are held by some other processes [21]. Data race happens when two or more threads access the same shared variable without proper synchronization, where at least one of the access is a write to the variable [22]. An atomicity violation occurs when an interleaved execution of a set of statements (expected to be atomic) by multiple threads is not equivalent to any serial execution of the same set of statements [23]. A serializable execution behaviour (scenario) is the one that always leaves the shared variables in consistent state.

As discussed in Shirole and Kumar [24], execution interleavings are divided into four classes: serialize, blocked, data-race, and non-data-race interleavings based on the execution behaviours that may uncover a specific class of concurrency errors. But, a simple path coverage criterion selects only one test scenario, which falls into any single class among the above classes of execution interleavings. Moreover, an activity path coverage criterion selects only one test scenario that falls either in blocked interleavings if thread are dependent or in no-data-race interleavings if threads are independent. Hence, selecting any simple interleaving path may not cover all the above concurrent behaviours. Moreover, there is no specific general thumb rule to select specific interleaving that addresses all the above classes. This demands more number of test scenarios for testing concurrent systems.

Although, concurrency has been addressed by the simple path coverage criterion. A complete distinction between sequential and concurrent behaviours, has not been considered. This distinction will help test management team to allocate separate resources towards sequential and concurrent behaviours for testing. Sequential behaviours will allow to have completed structural testing and the concurrent behaviours will allow to have concurrency testing separately. One may exclude the concurrency interleaving paths from the testing
schedule based on testing goals 80% coverage (stopping criteria) in rush delivery schedules. To avoid such slippage, there is need of distinction of the sequential and concurrent coverage criteria.

### 5 | PROPOSED CONCURRENCY COVERAGE CRITERIA

In an activity diagram, two types of execution paths are available: non-interleaving paths (which consist of activities, decisions, iterations, and merges) and interleaving paths (which consist of activities, decisions, iterations, merges, forks, and joins). Non-interleaving paths have a total order from the start activity to the end activity, and their execution represents a single trace. Interleaving paths have a partial order from the start activity to the final activity inside fork-join, and their execution represents different interleaving sequences. A basic path definition discussed by Linzhang et al. [7] does not distinguish between non-interleaving and interleaving paths for concurrent activity diagrams. Similarly, although simple path and activity path coverage criteria distinguish between non-interleaving and interleaving paths for concurrent activity diagrams, they still consider non-interleaving and interleaving paths under single coverage criterion. For concurrent activity diagrams, distinction between paths and their respective coverage criteria will help to effective effort estimation and test management for sequential and concurrency testing. Hence, we redefine interleaving and non-interleaving paths. Hence, we redefine basic path and basic path coverage criterion for only non-interleaving paths of activity diagrams. In addition, we also introduce new interleaving activity path and IAPC criterion for concurrent activity diagrams.

#### 5.1 | Proposed basic path coverage criterion

Traditional structural testing aiming for coverage of structural elements and sequential (non-concurrent) behaviours. To cover sequential behaviours and elements, we only consider activities, decisions, iterations, and merges of activity diagrams as non-concurrent constructs. Activity diagrams either include non-concurrent constructs and/or concurrency construct, that is fork-join. To eliminate ambiguous interleaving paths from structural testing, one should consider only non-concurrent constructs of activity diagrams. Thus, with this restriction basic path can be redefined as follows.

**Definition 2 Basic Path**: A path from the start activity to the end activity traversed by DFS method through non-concurrent constructs of an activity diagram, is called the basic path of the activity diagram, if all activities occurs once except some activities, like loop condition which occurs once or twice, in the path.

This revised new definition of basic path based on Linzhang et al. [7], which explores only non-interleaving paths of activity diagrams with or without concurrency constructs. Also satisfies Kundu and Samanta [17]’s condition of testing both the looping and non-looping paths for loop nodes. Thus, in basic path all activities barring the loop, should be visited once. In the loop condition, this needs to be visited once (in case of non-looping) or twice (in case of looping). Basic path coverage criterion based on this new definition of the basic path is defined as follows.

**Definition 3 Basic Path Coverage Criterion**: Let $P$ be the set of basic paths of an activity diagram, and $TS$ be the set of test scenarios generated from the activity diagram. The test set $TS$ satisfies basic activity path coverage, if and only if, $\forall p \in P, \exists t \in TS$ such that when the software is executed using $t$, the basic path $p$ of the activity diagram is executed.

The revised basic path coverage criterion is applicable for both non-concurrent activity diagrams and concurrent activity diagrams outside fork-join constructs. It includes all paths generated by DFS method for non-concurrent activity diagram. Whereas, it includes all paths generated by DFS method excluding concurrency construct parts of concurrent activity diagram.

For the motivating example in Figure 1, there are three basic paths that cover behaviours $B_1, B_2,$ and $B_4$ by the proposed basic path coverage criterion. The fourth behaviour $B_4$ is excluded as it includes a concurrency construct. The behaviour $B_4$ results in interleaving paths that is addressed by IAPC. The proposed basic path coverage criterion eliminates ambiguous interleaving paths and selects only sequential behaviours. Thus helping the test team on defining structural testing goals.

#### 5.2 | Interleaving activity path coverage criterion

Concurrent activity diagrams represent two types of the behaviours: parallel computing and concurrent computing. In parallel computing multiple small tasks that are independent to each-other execute simultaneously to complete a task. In this case, all threads are independent and their any execution interleaving will not result in concurrency errors except synchronization at join. In concurrent computing two or more threads executes simultaneously but they have some dependency among them that result in restricting some of the execution interleaving paths. Concurrent computing leads to concurrency errors if the implementation does not include required restrictions on some of the interleaving paths by imposing appropriate synchronization primitives. In a nutshell, multiple (independent) sub-tasks execute simultaneously to complete a task in parallel computing while multiple dependent tasks execute simultaneously to complete a task in concurrent computing, and this may lead to concurrency errors. Thus, for concurrent activity diagrams generating test set for concurrency error is a challenging task.
The simple path definition by Mingsong et al. [8] selects one representative path among all interleaving paths having the same set of activities and the same partial order relation. The simple path coverage criterion is suitable for parallel computing, where a single interleaving test scenario is good enough to test synchronization at join and the execution of all parallel tasks. However, this criterion is not sufficient for testing concurrency errors. This implies that for concurrency testing there is need of more number of test paths. In other words, a test suite should have at least one representative test scenario from each behaviour presented in the study by Shirole and Kumar [24] to uncover concurrency errors like data races, data inconsistency, and deadlocks. Besides these facts, it is not clear in advance which interleaving paths will lead to the specific concurrency behaviour; therefore, it is necessary to explore maximum number of the test scenarios for concurrency testing.

In code-based testing, Lu et al. [10] have proposed interleaving coverage criteria hierarchy for concurrent programs to systematically explore the interleaving space, and thereby effectively expose concurrency bugs. Similarly, we have proposed IAPC and TAPC criteria for UML-based testing derived from Lu et al.'s [10] coverage criteria hierarchy. We first define interleaving activity path and then IAPC criterion.

**Definition 4 Interleaving Activity Path**: A path from the start activity to the end activity through concurrent constructs of an activity diagram, is called the interleaving activity path if the path includes concurrent activity interleaving due to the fork-join construct and all activities occurs once except some activities, like loop condition which occurs once or twice, in the path.

Interleaving activity path occurs only in concurrent activity diagrams. These type of paths have three sub-paths: 1) a sequential prefix path, 2) a sequential suffix path, and 3) an interleaving sub-path, which has same set of activities and same partial order, between prefix and suffix sub-paths. For the motivating example in Figure 1, interleaving paths \( B_1^1, B_2^2, B_3^3, B_4^4, B_5^5, B_6^6 \) have 1) sequential prefix path \( S \rightarrow a_1 \rightarrow a_2 \rightarrow D_1 \rightarrow a_3 \rightarrow D_2 \rightarrow a_4 \rightarrow D_3 \rightarrow F \) and 2) sequential suffix path \( J \rightarrow a_{10} \rightarrow M_1 \rightarrow a_{11} \rightarrow E \); and activities \( \{a_{10}, a_{11}, a_6, a_9\} \) result in six different interleaving paths. IAPC criterion based on interleaving activity path definition is defined as follows.

**Definition 5 Interleaving Activity Path Coverage Criterion**: Let \( IP \) be the set of interleaving activity paths of an activity diagram, and \( TS \) be the set of test scenarios generated from the activity diagram. The test set \( TS \) satisfies interleaving activity path coverage, if and only if, \( \forall p \in IP, \exists t \in TS \) such that when the software is executed using ‘t’, the interleaving activity path \( p \) of the activity diagram is executed, fully or partially.

An IAPC criterion requires that all activity interleavings due to the fork-join construct of a concurrent activity diagram be covered. Some paths, which are blocked, execute partially as oppose to the other paths. For example, all interleaving paths \( B_1^1, B_2^2, B_3^3, B_4^4, B_5^5, B_6^6 \) are covered by the new IAPC criterion of Figure 1. This new coverage criterion covers only fourth behaviour \( B_4 \), which includes concurrency construct, whereas behaviours \( B_1, B_2, \) and \( B_3 \) are excluded. As compared with simple path IAPC criterion selects large number of test paths that will increase a chance of uncovering concurrency errors.

### 5.3 Total activity path coverage criterion

A non-concurrent activity diagram consists of only basic paths. A concurrent activity diagram consists of either interleaving activity paths and basic paths or only interleaving activity paths. Therefore, the TAPC criterion is proposed for concurrent activity diagrams which is a collection of both basic and interleaving paths.

**Definition 6 Total Activity Path Coverage Criterion**:

Let \( TP \) be the union of the set of basic paths and set of interleaving activity paths of an activity diagram, and \( TS \) be the set of test scenarios generated from the activity diagram. The test set \( TS \) satisfies total activity path coverage, if and only if, \( \forall p \in TP, \exists t \in TS \) such that the test scenario ‘t’ visits every activity and every transition of the path ‘p’ according to the order specified by \( p \).

In the case of non-concurrent activity diagrams, number of total activity paths is the same as the number of basic paths. TAPC criterion covers all paths of the Figure 1, that is, four behaviours represented by \( B_1, B_2, B_3, B_4 \), \( B_5, B_6, B_7, B_8 \), \( B_9, B_{10}, B_{11} \), \( B_{12}, B_{13}, B_{14} \), and \( B_{15} \) paths. In TAPC, both the non-interleaving and interleaving paths are combined. But complete coverage of the interleaving paths is not possible for large activity constructs, due to exponential increase in the number of paths. Thus, for exploring different concurrent behaviour, we need to find as many paths as possible. The scenarios including data races, data inconsistency and deadlocks are not known in advance, therefore higher coverage of interleaving paths may uncover some of these concurrency errors.

### 6 RESULTS AND DISCUSSION

We have analytically carried the coverage analysis of different types of concurrent activity diagrams. The test sets are generated from the activity diagrams. For this, first, an activity diagram is translated to an IR, a CFG-like structure, which we have named as ‘Activity CFG’. Then, we have generated test scenarios from the IR using the standard, though modified, DFS [25] and DFS-BFS [17] algorithms. Then, we have executed these test scenarios.
A modified DFS algorithm processes threads one after another inside fork-join construct thereby generating serializing test scenarios. In DFS-BFS algorithm, the DFS algorithm is used for non-concurrent parts of the activity graph, whereas the BFS algorithm is used for concurrent parts of the activity graph inside fork-join to generate test scenarios. The modified DFS and DFS-BFS algorithms run on the IR of activity diagram. An IR of a bank amount transfer transaction example (Figure 1) is shown in Figure 2. This IR is stored in a text file format for further processing, as shown in Figure 3.

We have considered four different applications with two threads each and different type of read and write access. Read and write access may lead to different behaviours based on the type of interleaving take place at run-time. Table 1 shows different application scenarios and their basic paths, interleaving paths, and their distribution in concurrent behaviours. The basic paths are calculated using the DFS algorithm for non-concurrent applications. The interleaving paths are calculated using the following equation:

\[
\left( \sum_{j=1}^{m} n_j \right)! / \prod_{j=1}^{m} (n_j!)
\]

where \( m \)—number of threads, \( n_j \)—number of nodes (activities) in thread ‘j’. For example, if there are two threads inside the fork-join construct with first thread having three nodes and second thread having two nodes, then total interleaving paths are \((3! + 2!)/(3!2!) = 5/3!2! = 10\). If any thread has a decision node inside the thread then calculation for that thread shall be done considering the true path and the false path separately.

i. **Airport check-in application**: It has no read and write access inside fork-join construct, hence all the execution paths represent the parallel computation and show non-data-race behaviour.

ii. **ATM money withdraw application**: It has read access inside fork-join construct. This read access is inside the decision node. The single read access leads to serializing behaviour and non-data-race behaviour.

iii. **ATM money transfer application**: It is both threads have read and write access inside fork-join construct. Both the threads access different data. The read and write access in both threads lead to serializing and blocking behaviors.

iv. **Order processing application**: It has read and write access by a single thread on different data. All read and write accesses by a single thread on different data leads to serializing and blocking behavior.

v. **Graphics utility application**: It has read and write access by both threads on the same data. In this application, the first thread has write access and the second thread has write access followed by two read accesses. All interleaving paths represent data-race, blocking, and non-data-race behaviors.

Due to limitations of the UML notations to show the synchronization primitives, (critical regions) in the UML syntax, we have inserted the synchronization primitives in the IR. Synchronization primitives of critical-region-start and critical-region-end are achieved by including lock and unlock activities around the activities, where data-access is required. Addition of the synchronization primitives in the IR will facilitate the concurrent behavior analysis. Therefore, we have added these nodes in the text file. We have shown IR of the motivating example, where only fork-join construct is explored, as shown in Figure 4.

The inclusion of the critical region information results in increase in the number of interleaving paths and a chance to explore different concurrent behaviours. Without including the above information, it would be difficult to carry out the experiments that represent results leading to different concurrency behaviours, as discussed in the study by Shirole and Kumar [24]. To understand effect of the inserting data access information to shared objects, we have carried out analysis. To do this analysis, we have chosen the application which have data access to the shared objects, that is, read and/or write. Table 2 shows synchronization primitives inserted to the number of threads, number of total paths, basic and interleaving paths, and the concurrent behavior as shown by the interleaving paths. Results show that insertion of the synchronization primitives increases the number of interleaving paths, at the same time, it facilitates to uncover the concurrent behaviours like serializing, blocking, data-race and no-data-race, as shown in Table 2. In short, addition of synchronization primitives explores the change in behaviour of the generated test scenarios.

If one does not include the synchronization primitive information to the shared data then resulting paths will not show
TABLE 1  Basic paths, Interleaving activity paths and their respective distribution in different concurrent behaviours [24] for application scenarios

| Application          | #Paths | Basic Paths | Interleaving Paths | Concurrent behaviours [24] |
|----------------------|--------|-------------|--------------------|--------------------------|
|                      |        |             |                    | Serializing | Blocking | Data-race | Non-data-race |
| Airport check In     | 8      | 1           | 7                  | –           | –        | –         | 8            |
| ATM money withdraw   | 13     | 3           | 10                 | 7           | –        | –         | 3            |
| ATM money transfer   | 73     | 3           | 70                 | 2           | 68       | –         | 3            |
| Order processing     | 422    | 2           | 420                | 420         | –        | –         | 2            |
| Graphics utility     | 792    | 0           | 792                | –           | 560      | 140       | 92           |

FIGURE 4  Addition of synchronization primitives (lock and unlock nodes) to different threads, (a) show no synchronization primitives, (b) synchronization primitives added to one thread, and (c) synchronization primitives added to both threads

TABLE 2  Effect of adding synchronization primitives (lock and unlock for critical region) information on number of paths and their distribution in different concurrent behaviors for activity diagrams

| Application          | Synchronization Primitives | #paths | Basic Paths | Interleaving Paths | Concurrent Behaviors [24] |
|----------------------|-----------------------------|--------|-------------|--------------------|--------------------------|
|                      |                             |        |             |                    | Serializing | Blocking | Data-race | Non-data-race |
| ATM money transfer   | No                          | 9      | 3           | 6                  | –           | –        | –         | 6            |
|                      | Single thread               | 18     | 3           | 15                 | –           | –        | 14        | 4            |
|                      | Both threads                | 73     | 3           | 70                 | 2           | 68       | –         | 3            |
| Order processing     | No                          | 32     | 2           | 30                 | –           | –        | –         | 32           |
|                      | Single thread               | 422    | 2           | 420                | 420         | –        | –         | 2            |
| Graphics utility     | No                          | 56     | 0           | 56                 | –           | –        | 56        | –            |
|                      | Single thread               | 252    | 0           | 252                | –           | –        | 200       | 52           |
|                      | Both threads                | 792    | 0           | 792                | –           | 560      | 140       | 92           |
any concurrent behavior, and test manager will not get complete idea of the possible risk in the future. Test team may only work on structural and parallel computation testing.

To clarify, how the number of interleaving paths increase, we have demonstrated the calculation of number of paths for the ATM Money Transfer example. We have considered three cases as shown in Figure 4: the number of interleaving paths, in case, (a) no insertion of synchronization primitives: \((4!/2!2!) = 6\), (b) with synchronization primitives in single thread: \((6!/4!2!) = 15\), and (c) with synchronization primitives in the both thread: \((8!/4!4!) = 70\). Similarly, the calculation was carried out for other applications, as shown in Table 2. The analysis of the generated test scenarios is carried out using the approach presented in the study by Shirole and Kumar [24].

For ATM Money Transfer and Graphics Utility application scenarios, all the interleaving paths are categorized as data-race behavior. Similarly, all the interleaving paths are categorized as non-data-race behavior for the Order Processing application scenarios. The effect of inserting synchronization primitives activities, that is, critical-region-start (lock) and critical-region-end (unlock), in a single thread and both threads, are shown in Table 2. For the Order Processing application scenarios, only one thread has shared data access, therefore the analysis is carried out only up to a single thread, and the resulting interleaving paths are classified as serializing behavior. For the ATM Money Transfer and the Graphics Utility application scenarios, both the threads have shared data access, hence the analysis is carried for a single and two threads. For the ATM Money Transfer application scenario, inserting synchronization primitives to the single thread does not give any additional information. But, inserting synchronization primitives to both the threads result in exploring concurrent behaviors, namely serializing, blocking and non-data-race. Similarly, inserting synchronization primitives to both the threads result in exploring concurrent behaviors, namely blocking, data-race and non-data-race for Graphics Utility application scenarios.

We have carried out coverage analysis of the application scenarios discussed above for the existing and the proposed coverage criteria based on the test suite generated by the modified DFS [25] and the DFS-BFS [17] algorithms. Table 3 shows the result of percentage coverage results for the existing coverage criteria, namely basic path and simple path, and proposed coverage criteria, namely basic path, interleaving activity path, and total activity path. Total four test scenarios are generated by the modified-DFS and DFS-BFS algorithms for ATM Money Transfer Transaction application. For clarity, we have shown only the part inside the fork-join construct based on the IR, as shown in Figure 4. The test scenarios generated by the modified DFS algorithm are:

\[
TS_1 = F \xrightarrow{t_{i2}} lock \xrightarrow{t_{i3}} a_6 \xrightarrow{t_{i4}} a_7 \xrightarrow{t_{i5}} unlock \xrightarrow{t_{i6}} CS \xrightarrow{t_{i7}} lock \xrightarrow{t_{i8}} a_5 \xrightarrow{t_{i9}} unlock \xrightarrow{t_{i10}} CS \xrightarrow{t_{i11}} J.
\]

\[
TS_2 = F \xrightarrow{t_{i7}} lock \xrightarrow{t_{i9}} a_8 \xrightarrow{t_{i10}} d_5 \xrightarrow{t_{i12}} unlock \xrightarrow{t_{i13}} CS \xrightarrow{t_{i14}} lock \xrightarrow{t_{i15}} a_6 \xrightarrow{t_{i16}} a_7 \xrightarrow{t_{i17}} unlock \xrightarrow{t_{i18}} CS \xrightarrow{t_{i19}} J.
\]

The modified DFS algorithm generates two paths with context-switch at end of the thread. The test scenarios generated by DFS-BFS algorithm are:

\[
TS_3 = F \xrightarrow{t_{i2}} lock \xrightarrow{t_{i3}} CS \xrightarrow{t_{i4}} lock \xrightarrow{t_{i5}} d_5 \xrightarrow{t_{i6}} unlock \xrightarrow{t_{i7}} CS \xrightarrow{t_{i8}} J.
\]

\[
TS_4 = F \xrightarrow{t_{i7}} lock \xrightarrow{t_{i9}} CS \xrightarrow{t_{i10}} a_8 \xrightarrow{t_{i11}} unlock \xrightarrow{t_{i12}} CS \xrightarrow{t_{i13}} J.
\]

DFS-BFS algorithm generates two paths with a context-switch after every node in a path.

Results in Table 3 show that the modified DFS algorithm [25] and the DFS-BFS algorithm [17] achieve 100% basic path and simple path coverage, whereas these two algorithms achieve marginal IAPC. Both the approaches have generated all basic paths, and thus achieve 100% basic path coverage. The modified DFS algorithm generated two serializing interleaving paths for each application scenario, thereby uncovers serializing and non-data-race concurrent behaviors. The DFS-BFS algorithm generated two interleaving paths for each application scenario, thereby uncovers blocking and non-data-race behaviors. In case of Graphics Utility application, we have considered scenario inside fork-join construct only. Thus, there is no non-concurrent paths, as mentioned ‘not applicable’ (NA) in Table 3 for the basic path. Results show that for all types of applications the number of interleaving paths generated are same by modified DFS and DFS-BFS algorithms.

Based on the above results, it is clear that a simple path is good enough criterion for the parallel computations and applications where there is only read or write access by a single thread. In contrast, when more than one thread accesses data for read and/or write access, then simple path coverage criteria selects a test scenario probably from the behaviour whose the path count is maximum. For Graphics Utility application, non-data-race behaviour is having high number of test scenarios, hence simple path may select a path that exhibit non-data-race behaviour and other behaviours where actual problem of concurrency error is not uncovered. Thus, to uncover all such hidden behaviours by the execution interleaving, one has to explore maximum number of test scenarios that cover different concurrent behaviours. Therefore, the proposed coverage criteria increase a chance of selection of different test scenarios of different concurrent behaviours. It allows to select a maximum number of test scenarios under the concurrency setting. In fact, 100% coverage achievement may not be possible, but generated test scenarios show correct classification of the behaviour, thus such a representative of each behaviour may be enough for testing purpose.

The coverage criteria for concurrent design in the form of activity diagrams with fork-join constructs are focused. We have further evaluated the proposed concurrent coverage criteria in terms of completeness, applicability, usability, runtime performance, and limitations, as follows:

- **Completeness**: (a) basic path coverage; (b) completeness is possible with the help of existing test scenario generator;
(b) interleaving activity path coverage: completeness is not possible due to exponential test scenarios. However, selecting test scenarios that satisfy the concurrency behaviours discussed by Shirole and Kumar [24], will give concurrency behaviour completeness. To achieve this maximum number of paths need to be generated which may fall in different concurrency behaviours; (c) total activity path coverage: it is combined coverage of both basic path and interleaving activity path coverage, hence complete coverage is not possible because interleaving activity path coverage is not possible to achieve completely.

**Applicability:** (a) basic path coverage: it is applicable for the both concurrent and non-concurrent application designs; (b) interleaving activity path coverage: it is only applicable for concurrent application designs; (c) total activity path coverage: it is applicable for the both concurrent and non-concurrent application designs.

**Usability:** (a) basic path coverage: it is usable for the both concurrent and non-concurrent application designs; (b) interleaving activity path coverage: it is only usable for concurrent application designs; (c) total activity path coverage: it is usable for the both concurrent and non-concurrent application designs.

**Runtime Performance:** (a) basic path coverage: for this coverage, test scenarios are generated using existing DFS and DFS-BFS algorithms, hence the run-time performance $O(V + E)$, which is good; (b) interleaving activity path coverage: for this coverage, test scenarios are generated using existing DFS and DFS-BFS algorithms, but number of test scenarios are not sufficient. A new algorithm need to be considered, which may explore the interleaving space. As interleaving space is exponential and the run-time performance is not good in this case; (c) total activity path coverage: it is combined coverage of both basic path and interleaving activity path coverage. So run-time performance is good in case of non-concurrent activity design and concurrent activity design with no data access in any thread; and not good in case of concurrent activity design with data access on both threads.

**Limitations:** (a) basic path coverage: no limitations as there are existing test scenario generators achieve 100 % basic path coverage; (b) interleaving activity path coverage: currently there is no possibility of achieving 100 % interleaving path coverage with existing test scenario generators. Need to explore on test scenario generator that will increase the coverage; (c) total activity path coverage: no limitation for non-concurrent activity design, however for concurrent activity designs there are some limitations.

Some of the above issues are open research areas in concurrent system design.

## 7 CONCLUSION

Clear distinction between sequential and concurrent execution behaviours of concurrent activity diagrams is indicated. To take account of distinct execution behaviours, we have redefined basic path coverage criterion, and presented new coverage criteria, namely interleaving activity path and TAPC for concurrent activity diagrams. IAPC criterion includes more interleaving test scenarios as compared to the simple path coverage criterion. The proposed coverage criteria satisfy concurrent behaviours, namely serializing, synchronization, blocking, and non-blocking. Thereby, the above proposed criteria increase the probability of uncovering concurrency errors. Distinction of non-interleaving and interleaving paths in concurrent activity diagrams helps to effective effort estimation and test management for sequential and concurrent testing. The future work may explore larger number of test scenarios that may cover different concurrent behaviours.

## ACKNOWLEDGMENTS

The authors thank the reviewers for their insightful comments and suggestions. The authors are grateful to one anonymous reviewer who has very precisely summarized the whole work in a paragraph, which the authors have included in the Section 1. The authors express their gratitude to the Editor-in-Chief, the Associate Editor, and Editorial Office Assistant(s) of this journal for managing this manuscript. The authors acknowledge that the major part of this work was done while both of us were at Indian Institute of Technology Kharagpur.
REFERENCES

1. Souza, S.R.S., Brito, M.A.S., Silva, R.A., Souza, P.S.L., Zaluska, E.: Research in concurrent software testing: a systematic review. In: Proc. Workshop Parallel & Distributed Systems: Testing, Analysis & Debugging, pp. 1–5. (ACM Press) (2011)

2. Kamonsantitiroj, S., Pipanmaekaporn, L., Lorpuunmanee, S.: A memorization approach for test case generation in concurrent UML activity diagram. In: Proc. 2nd Int. Conf. GeoInformatics & Data Analysis, pp. 20–25. (ACM Press) (2019)

3. Yimman, S., Kamonsantitiroj, S., Pipanmaekaporn, L.: Concurrent test case generation from UML activity diagram based on dynamic programming. In: Proc. 6th Int. Conf. Software & Computer Applications, pp. 33–38. (ACM Press) (2017)

4. Lima, L., Tavares, A., Nogueira, S.C.: A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP. Sci. Comput. Program. 197, 102497 (2020)

5. Beizer, B.: Software testing techniques, 2nd ed. Van Nostrand Reinhold Co., NY (1990)

6. Ammann, P., Offutt, J.: Introduction to software testing, 1st ed. Cambridge University Press, NY (2008)

7. Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., Guoliang, Z.: Generating test cases from UML activity diagram based on gray-box method. In: Proc. 11th Asia-Pacific Software Engineering Conference, pp. 284–291. (IEEE) (2004)

8. Mingsong, C., Xiaokang, Q., Xuandong, L.: Automatic test case generation for UML activity diagrams. In: Proc. Int. Workshop Automation of Software Test, pp. 2–8. (ACM Press) (2006)

9. Shirole, M., Kumar, R.: UML behavioral model based test case generation: a survey. ACM SIGSOFT Softw Eng Notes. 38(4), 1–13 (2013)

10. Lu, S., Jiang, W., Zhou, Y.: A study of interleaving coverage criteria. In: Proc. 6th Joint Meeting European Software Engineering Conf. and ACM SIGSOFT Symp. Foundations Software Engineering, pp. 533–536. (ACM Press) (2007)

11. Weißleder, S.: ‘Test models and coverage criteria for automatic model-based test generation with UML state machines’ [Doctoral thesis], Humboldt-University Berlin, Germany (2010)

12. Bron, A., Farchi, E., Magid, Y., Nir, Y., Ur, S.: Applications of synchronization coverage. In: Proc. 10th ACM SIGPLAN Symp. Principles and Practice Parallel Programming, pp. 206–212. (ACM Press) (2005)

13. Kim, H., Cho, D., Moon, S.: Maximizing synchronization coverage via controlling thread schedule. In: Proc. Consumer Communications and Networking Conf, pp. 1186–1191. (IEEE) (2011)

14. Abdurazik, A., Offutt, J.: Using UML collaboration diagrams for static checking and test generation. In: Proc. 3rd Int. Conf. UML: Advancing Standard, pp. 383–395. (Springer-Verlag) (2000)

15. Ferreira, R.D.F., Faria, J.P., Paiva, A.C.R.: Test coverage analysis of UML state machines. In: Proc. 3rd Int. Conf. Software Testing, Verification, and Validation Workshops, pp. 284–289. (IEEE) (2010)

16. Kosmatov, N., Legeard, B., Peureux, E., Utting, M.: Boundary coverage criteria for test generation from formal models. In: Proc. 15th Int. Symp. Software Reliability Engineering, pp. 139–150. (IEEE) (2004)

17. Kundu, D., Samanta, D.: A novel approach to generate test cases from UML activity diagrams. J. Object. Technol. 8(5), 65–83 (2009)

18. Routnev, A., Kagan, S., Sawin, J.: Coverage criteria for testing of object interactions in sequence diagrams. In: Fundamental Approaches to Software Engineering. vol. 3442 of LNCS, pp. 289–304. Springer) (2005)

19. Dinh-Trong, T.T., Ghosh, S., France, R.B.: A systematic approach to generate inputs to test UML design models. In: Proc. 17th Int. Symp. Software Reliability Engineering, pp. 95–104. (IEEE) (2006)

20. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. SIGPLAN Not. 26(12), 85–96 (1991)

21. Naik, M., Park, C.S., Sen, K., Gay, D.: Effective static deadlock detection. In: Proceedings of the 31st International Conference on Software Engineering. ICSE ’09, pp. 386–396. IEEE Computer Society, Washington (2009)

22. Savage, S., Burns, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans Comput Syst. 15, 391–411 (1997)

23. Farzan, A., Madhusudan, P.: Causal atomicity. In: Ball, T., Jones, R. (eds.) Computer Aided Verification. Vol. 4144 of Lecture Notes in Computer Science, pp. 315–328. Springer Berlin Heidelberg (2006)

24. Shirole, M., Kumar, R.: Test scenario selection for concurrency testing from UML models. In: Proc. 8th Int. Conf. Contemporary Computing, pp. 531–536. (IEEE) (2015)

25. Chandler, R., Lam, C.P., Li, H.: ‘AD2US: An automated approach to generating usage scenarios from UML activity diagrams’. In: Proc. 12th Asia-Pacific Software Engineering Conf, pp. 9–16. (IEEE) (2005)

How to cite this article: Shirole M, Kumar R. Concurrency coverage criteria for activity diagrams. IET Softw. 2021;15:43–54. https://doi.org/10.1049/ sfw2.12009