A Novel Approach for Deriving Test Scenarios and Test Cases from Events

Sandeep K. Singh*, Sangeeta Sabharwal** and J.P.Gupta***

Abstract—Safety critical systems, real time systems, and event-based systems have a complex set of events and their own interdependency, which makes them difficult to test ma Safety critic Safety critical systems, real time systems, and event-based systems have a complex set of events and their own interdependency, which makes them difficult to test manually. In order to cut down on costs, save time, and increase reliability, the model based testing approach is the best solution. Such an approach does not require applications or codes prior to generating test cases, so it leads to the early detection of faults, which helps in reducing the development time. Several model-based testing approaches have used different UML models but very few works have been reported to show the generation of test cases that use events. Test cases that use events are an apt choice for these types of systems. However, these works have considered events that happen at a user interface level in a system while other events that happen in a system are not considered. Such works have limited applications in testing the GUI of a system. In this paper, a novel model-based testing approach is presented using business events, state events, and control events that have been captured directly from requirement specifications. The proposed approach documents events in event templates and then builds an event-flow model and a fault model for a system. Test coverage criterion and an algorithm are designed using these models to generate event sequence based test scenarios and test cases. Unlike other event based approaches, our approach is able to detect the proposed faults in a system. A prototype tool is developed to automate and evaluate the applicability of the entire process. Results have shown that the proposed approach and supportive tool is able to successfully derive test scenarios and test cases from the requirement specifications of safety critical systems, real time systems, and event based systems

Keywords—Events, Event Meta Model, Testing, Test cases, Test scenarios, Event Based Systems, Software Engineering

1. INTRODUCTION

Event based systems ranging from real time monitoring systems in production, logistics and networking to complex event processing in finance and security are rapidly gaining importance in many application domains. The event based paradigm has gathered momentum as witnessed

Manuscript received April 27, 2011; first revision February 1, 2012; accepted March 19, 2012.
Corresponding Author: Sandeep K. Singh
* Department of Computer Science and Engineering and Information Technology, JIIT University, A-10 Sector 62, Noida, INDIA (sandeepsandy6@rediffmail.com, sandeepk.singh@jiit.ac.in)
** Division of Computer Science and Engineering, NSIT. Sector-3 ,Dwarka, New Delhi, INDIA
*** Sharda University, Plot No.32-34, Knowledge Park III, Greater Noida, INDIA

Copyright ⓒ 2012 KIPS (ISSN 1976-913X)

by current efforts in areas including event driven architectures, complex event processing, business process management and modeling, grid computing, web services notifications, event stream processing, and message-oriented middleware [1-16, 21-26]. The increasing popularity of event based systems has opened new challenging issues for event based systems. One such issue is testing the functional requirements of event based systems.

Manual Testing is a very costly and time consuming process. In order to cut down on costs, save time, and increase reliability, model based testing approaches are used. Model Based Testing (MBT) is a process of generating test cases and evaluating test results based on the design and analysis models. MBT lies in between specification and code based testing, hence it is also referred to as a gray box testing approach. A wide range of models like UML, SDL, Z, state diagrams, data flow diagrams, control flow diagrams, etc. have been used in MBT [33-57], whereas very few works [17 - 20] have been reported to show the generation of test cases using events. However, these works have taken into account events that happen at a user interface level in a system while other events that happen in a system are not taken into account in these bodies of work. Such works have limited applications in testing the GUI of a system. Thus, the motivation of this research is to use other types of events from specifications of a system and automate test case and test scenario generation.

In this paper, a novel model-based testing approach is presented using business events, state events, and control events that have been captured directly from requirement specifications. The proposed approach documents events in event templates and then builds an event-flow model and a fault model for a system that is being tested. Test coverage criterion and an algorithm are designed by using these models to generate event sequence based test scenarios and test cases. Unlike other event based approaches, our approach is able to detect the proposed faults in a system. A prototype tool has been developed to automate and evaluate the applicability of the entire process. Results have shown that the proposed approach and its supportive tool are able to successfully derive test scenarios and test cases from the requirement specifications of safety critical systems, real time systems, and event based systems.

In order to generate test cases and test scenarios using events from specifications, the following issues are identified:

(a) How can possible relationships and dependencies be represented among events occurring in an event based system ?
(b) Which model is appropriate for the Model Based Testing of event-based systems?
(c) What are the possible causes for faults due to complex relationships and dependencies among events in an event-based system?
(d) what algorithm is to be designed to generate test scenarios that not only gives appropriate coverage of the model but also detects causes of faults in an event-based system ?
(e) What should be augmented in an existing approach so that test cases are also generated in an event-based system?
(f) How to automate the entire process, so as to save time involved in the analysis and testing of event-based systemS ?
(g) How to evaluate and validate the proposed approach,in order to judge its usefulness?

Following contributions are made to address the above mentioned issues:

(a) Various event interdependency operators are defined to represent relationships among events. These relationships are described as Event-Flow Interdependency.
(b) An Event-Flow model is designed based on the Event-Flow Interdependency that has been identified. This abstract model is stored as an Event-Flow Graph (EFG).
(c) A Fault Model is defined by using the EFG to show the possible causes for faults due to complex relationships and dependencies among events.
(d) An algorithm is designed using a combination of classical breath-first and depth-first searches, which generate test scenarios and test cases from the EFG. Our algorithm is also capable of detecting faults like synchronization faults, loop faults, and events occurrence faults. Such faults are not addressed in the existing event based test case generation approaches.
(e) A process is described to annotate the Event-Flow Graphs with the necessary test information. This information is used at the level of testing individual events. A data structure, called an Event node Description Table (EnDT) is described in section 4.3.
(f) A prototype tool has also been developed to automate and validate the entire process of generating an EFG from elementary events and for computing test scenarios and test cases using the proposed approach.
(g) Several case studies are used as the input for the prototype tool. An analysis of one case study named the, "Automatic Production Environment (APE)" for a real time system that was taken from the Real-Time Lab of Embry-Riddle Aeronautical University [27] is shown in detail throughout the paper, whereas the results of others are show in tabular form. Results have shown that the proposed approach and its supportive tool can be of great help for testing safety critical systems, real time systems, and event based systems.

The organization of our paper is as follows: Section 2 discusses related research on model based testing approaches using various UML models as well as events. Section 3 details the proposed methodology to generate test scenarios and test cases using events along with its detailed application to a case study. Section 4 discusses the evaluation of the proposed approach for a real time application. Section 5 discusses tool support and the results of the evaluation on various cases and finally, Section 6 describes the conclusion and the future work.

2. Comparisons with Related Bodies of Work

A wide range of models like UML, SDL, Z, state diagrams, data flow diagrams, control flow diagrams, etc. have been used in MBT [33-57]. In [33], conformance test cases are generated for a communication protocol, which were modeled in an EFSM (Extended Finite State Machine) by a fault coverage analysis. In [34], authors have described the Finite State Machine (FSM) based test case generation, the fault models for test case generation, fault coverage, and three methods for test case generation using FSMs. Several approaches have used activity diagrams [35-42]. In [35], the authors have generated test cases based on activity path coverage criterion to cover faults like synchronization faults and faults in a loop. In [36], test cases are generated by comparing program execution traces with the given activity diagram. Test cases in [37] have been generated based on an I/O Explicit Activity Diagram (IOAD) model. In [38], a gray-box method is used to generate test cases. In [39], the authors have developed an automated tool

called TSGAD (Test Scenarios Generator for Activity Diagrams), which uses adaptive agents to directly generate test scenarios. Work in [40] introduced an approach that capture, store and output, usage scenarios that have been automatically derived from UML activity diagrams. In [41], the authors have used anti-ant like agents to directly generate test threads from the UML artifacts. Work in [42] used specification coverage to generate properties as well as design models to enable directed test generation using model checking.

Approaches using sequence diagrams are described in [43 - 49]. In [43], authors have created Message Dependence Graphs (MDG) from UML sequence diagrams for the generation of test cases. Test cases are generated in [44] by using both sequence diagrams and state diagrams. In [45], the combination of TTCN- 3, as the test description language, and UML Message Sequence Charts (MSC) are used to specify and automatically generate test cases for communication protocols and distributed systems. Authors in [46] have used OCL to generate test cases automatically from UML sequence diagrams. Authors in [47] have generated test cases by augmenting the sequence diagram graph (SDG) nodes with different information from use case templates, class diagrams, and the data dictionary to compose test vectors. In [48], the testing of mobile phone applications is done by translating UML sequence diagrams into Labeled Transition Systems (LTSs). In the work for [49], the authors generated test cases by using a formal operational semantics for a sequence.

Some of the model based testing approaches have used state diagrams and state charts [50 - 52]. In [50, 51] test cases are generated based on control and data flow in UML state diagrams. Work in [52] generates test cases from a flattened hierarchy structure, which is called the Testing Flow Graph (TFG). A few of the works have reported use of more than one specific UML model and have taken benefits from the features of each of the models involved [53-57]. In [53] activity diagrams are used to generate test scenarios and then class diagrams of each scenario are used to achieve maximum path coverage criteria. Work in [54] has used UML state machines, UML class diagrams, and OCL expressions to generate test cases. In [55], both use case and state diagrams are used for generating system-level test cases. In [56] authors have transformed Use Cases to a UML state model to generate test cases. Finally, work in [57] used an AI (Artificial Intelligence) planner to generate test cases from test objectives, which were derived from UML class diagrams.

Very few works [17-20] have been reported to show the generation of test cases using events. The role of events in testing is found in the area of GUI testing. A scalable and reusable model (called the event-flow model) of GUIs based on event interactions is presented in [17]. Combined with customized Event-Space Exploration Strategies (ESESs), the proposed event-flow model is used to quickly generate a large number of GUI test cases, which are effective in detecting GUI faults. Event-based notions and tools are also used to generate and select test cases systematically and introduce a holistic view of fault modeling [20]. A new GUI automation test model is presented in [19] that is based on an event-flow graph model to generate test cases in the daily smoke test and to create test cases in a deep regression test. Work in [18] has used events and event sequences to define coverage criteria for GUIs to help determine whether a GUI has been adequately tested.

However, these works are limited to only the GUI testing of a system. A GUI is a hierarchical, graphical front-end for a software system that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. The events associated with a GUI are functions performed on different types of objects, in different

contexts, yielding different behavior (i.e., load, select, play, remove, shift right/left, cutter off/on, pressure off/on etc.) They have made a GUI events based model and are able to detect GUI related faults. But other than these GUI events, there are other important non-GUI events that happen in a system like state events, control events, and external events. These works have not factored in such events. Thus contribution to this research cannot be compared with work done in the event-based testing of GUI, as the two event perspectives are totally different. Earlier works focus on GUI events, whereas work done in this paper has considered non-GUI that is related with functional requirements. These works have to rely on a commercial capture playback tool like GUI Ripper that uses reverse-engineering techniques to automatically construct the event-flow graphs and integration tree. During "GUI Ripping," the GUI application is executed automatically and the application's windows are opened in a depth-first manner. The GUI Ripper extracts all of the widgets and their properties from the GUI. These attributes are used to construct the event-flow graphs and integration tree. But the problem is that the GUI Ripper is not perfect, (i.e. parts of the retrieved information may be incomplete/incorrect). Tools have been developed that may be used to manually "edit" the event-flow graphs and integration tree and fix these problems. These tools involve tedious and error-prone manual work. In comparison to these approaches, the work presented in this paper differs in terms of the following aspects: (1) none of the techniques/tools have worked on requirement specifications directly and they either need complete GUI along with a capture replay tool or an adjacency matrix that relates to various GUI events, whereas we have extracted events directly from requirements. (2) The work presented in these articles relies on capture replay tool to generate the event flow diagram, whereas we have devised an algorithm to do the same thing automatically. (3) These approaches have used GUI events, whereas we have used state events, control events and external events related with the functional requirements of a system. Earlier works have focused on GUI faults (i.e., faults when the GUI does not perform as documented in the specifications and/or design or faults in which GUI shows unacceptable software behavior for which there may be no explicit specifications or design for things like software crashes and screen "freezing"), whereas our fault model deals with event interaction faults due to a problem with the event interdependency operators. Thus, faults detected by earlier approaches based on GUI are different from faults detected in our approach and hence cannot be compared.

Unlike UML models based testing approaches, we have used events and their relationships instead of UML models to build a model for an event-based SUT, as UML models like use case, activity, or sequence diagrams are use case oriented models and are not based on events. Use cases are not events; rather they are end results of event triggers. Other UML models like class and state diagrams focus on a testing component (i.e., on class rather than an entire system), whereas our approach works at testing the overall system specifications.

3. PROPOSED METHODOLOGY

This section discusses our proposed approach to generate test scenarios and test cases from events. Our approach consists of the following four steps and each step is demonstrated along with its application to the APE case study [27]:

1. Extract and document events from requirements using the proposed Event Templates.
2. Generate Event-Flow Interdependencies from the Event Templates.
3. Generate an Event-Flow Model in the form of an Event-Flow Graph.
4. Find possible causes of faults in an Event-Flow Graph
5. Generate Event Sequence-Based Test Scenarios from an Event-Flow Graph.
6. Generate test cases by augmenting Event-Flow Graphs with the necessary test information

3.1 Extracting and documenting events occurring in a system using the proposed Event Templates

Unlike other event based test case generation approaches, we have used business events, state events, and control events that have been captured directly from requirement specifications to generate test scenarios and test cases. For example, a customer places an order and then the customer sends their payment. In this case, the arrival of an order or a payment in a system indicates that flow-oriented events have occurred. Temporal events are triggered by the arrival of a point in time. For example, a customer needs a (monthly) statement. Accounting needs (daily) cash receipts. Control events are events that occur when something happens in a system and the system must initiate some process in response to this event. For example, a book reprint order arrives at the warehouse as result of a drop in the number of copies of a book.

Safety critical systems, real time systems, and event-based systems have a complex set of events and interdependency. It is very difficult to identify, as well as document events and their interdependency, manually. So in our approach, events are identified and extracted from the requirements automatically by using a Subject, Verb, and Object (SVO) pattern. An SVO pattern identifies an event in a sentence (i.e., events in a typical order processing system are as follows: "Customer places order," "Sales Manager denies credit request," "Marketing Department changes prices.") The extraction process takes natural language textual requirements written in English as input and gives output in a textual format, as shown in Fig. 1.

3.2 Event-Flow Interdependency from Event Templates

This sub-section addresses the issue of representing possible relationships and dependencies among events occurring in an event-based system. In order to do this, various event interdependency operators are defined to represent relationships among events. These relationships are described as Event-Flow Interdependency. The procedure for finding the Event-Flow Interdependency from Event Templates is discussed below.

In the construction of an Event-Flow Model, the first step is to find the Event-Flow Interdependency. Events in a system do not occur in isolation but as a chain of events. So, the identification of an event in turn identifies other events. This relationship among events is described as Event-Flow Interdependency. An event can trigger either a single event, or a set of events that can be executed independently or in parallel. Similarly, either one or more events can be causative events for an event [3]. Events are related with one another using connector nodes like "split," "join," and "not." A "split" is a connector node that divides into two or more transitions leading to different events. A "join" is a connector node that joins, unites, or links transitions from different events. A "not" connector node represents the non-occurrence of an event. It indicates that an event can be a negation of another event. (e.g. for an event "IR sensor detect the boundary," negation would be "IR sensor does not detect the boundary" . These three connector

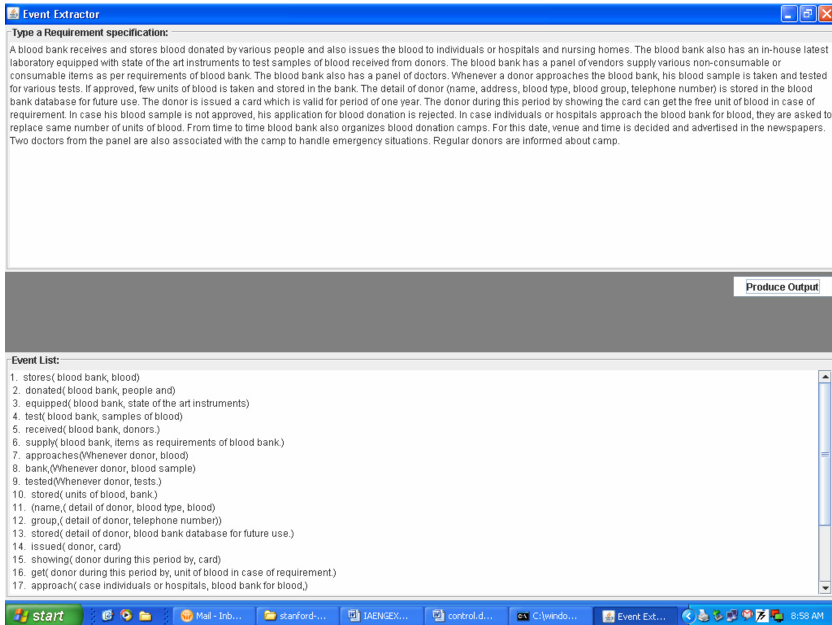


Fig. 1. Event Extractor Module

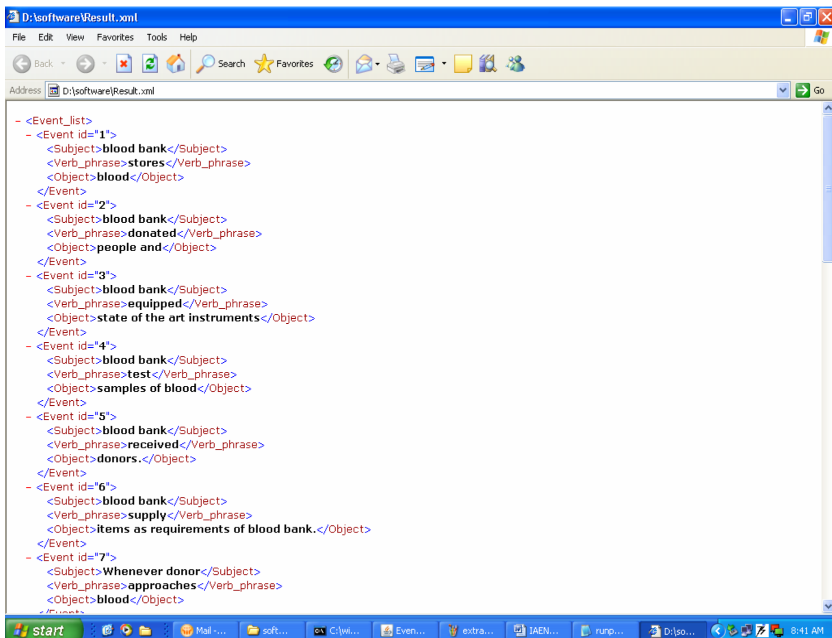
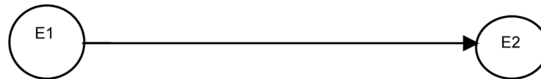


Fig. 2. Output of an Event Extractor Module in XML format

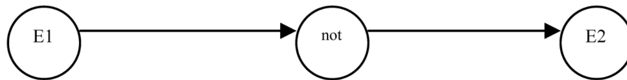
nodes combined with the "and," "xor," or "or" logical operators form different possible event interdependencies. An Event-Flow Interdependency is captured from causative events and the

trigger vector section of an Event Template. Fig. 3 depicts all of the possible Event-Flow Interdependencies in a system.

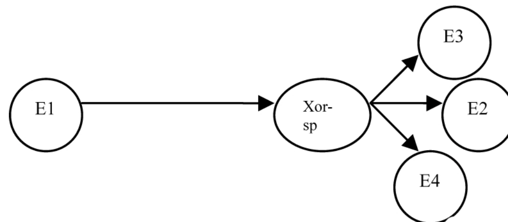
(a) Causative events can be a single event or a set of events that are related, which are using the “join” connector node. A connector node with more than one incoming transition is classified as an Event-and join (join), an Event-or join (or-j), and an Event-xor join (xor-j),



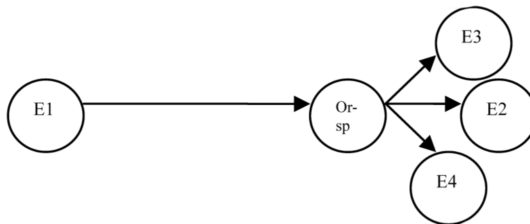
(a) E1 is causative event of E2 and E2 is a trigger event of E1 with a simple dependency



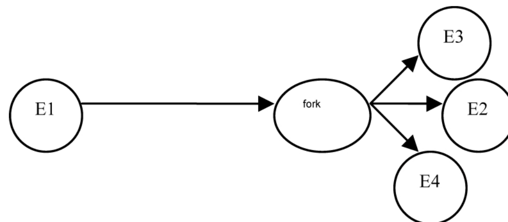
(b) E1 is causative event of E2 and E2 is a trigger event of E1 with a not (non-event) dependency



(c) E1 is causative event of E2, E3, and E4 and E2, E3, and E4 are trigger events of E1 with a xor –split dependency

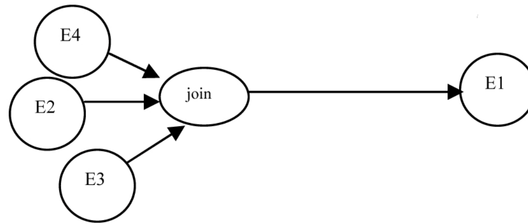


(d) E1 is causative event of E2, E3, and E4 and E2, E3, and E4 are trigger events of E1 with an or-split dependency

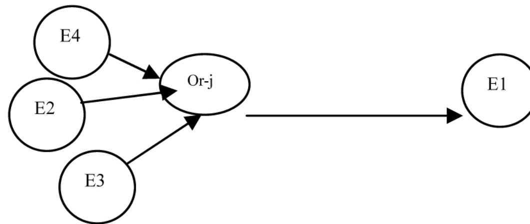


(e) E1 is causative event of E2, E3, and E4 and E2, E3, and E4 are trigger events of E1 with a fork (and-split) dependency

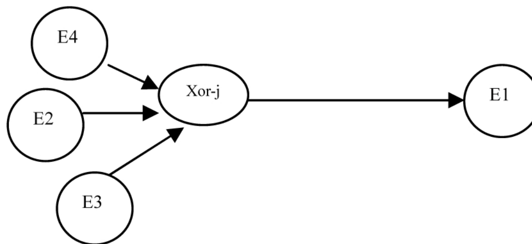
Fig. 3. Basic Event-Flow Interdependencies in a system



(f) E1 is trigger event of E2, E3, and E4 and E2, E3, and E4 are causative events of E1 with a join (and-join) dependency



(g) E1 is a trigger event of E2, E3, and E4 and E2, E3, and E4 are causative events of E1 with an or-join dependency



(h) E1 is a trigger event of E2, E3, and E4 and E2, E3, and E4 are causative events of E1 with an xor-join dependency

Fig. 3. Continue

as shown in Figs 3(c), 3(d), and 3(e). An event following an event-and join node starts its execution when all its incoming events are executed.

(b) Events triggered in a trigger vector are related with triggering events using the "split" connector node. A connector node with more than one outgoing transition is classified as an Event-and split (fork), an Event-or split (or-sp), or an Event-xor split (xor-sp), as shown in Figs 3(f), 3(g), and 3(h). An Event-and split indicates that all events have to be triggered in parallel. Fig. 3(a) represents simple event interdependency, whereas Fig. 3(b) represents an event with a "not" node.

3.3 Generate an Event-Flow Model in the form of an Event-Flow Graph

This sub-section addresses the issue of constructing an appropriate model for testing event-based systems. An Event-Flow Model is designed based on the Event-Flow Interdependency, which was identified earlier. This abstract model is stored as an Event-Flow Graph (EFG). The procedure to generate an Event-Flow Model and an Event-Flow Graph is discussed below.

An Event-Flow Model as an Event-Flow Graph All events happening in the system are represented collectively using the Event-Flow Model. In much the same way as a control-flow model represents all of the possible execution paths in a program [28] and a data-flow model represents all of the possible definitions and uses of a memory location [29] the Event-Flow Model represents all of the possible sequences of events that can be executed in a system. The Event-Flow Model contains two parts. The first part refers to events that are causes in terms of causative events and the second part refers to events that are triggered after an event has been executed. Both of these parts play an important role in constructing an Event-Flow Model. An Event-Flow Model is represented as an Event-Flow Graph.

An Event-Flow Graph (EFG) represents events, their interaction in terms of causative events, and the trigger vector. It may represent an infinite looping condition where there is no exit or it may have one event designated as a start event and another designated as an exit event. A start event indicates the initialization of a system (i.e., once a system is turned on, all other events can be executed). An exit event indicates the completion of functionality. In an Event-Flow Graph, events are represented using circles and dependencies among events and are represented using arrows. The proposed Event-Flow Graph is different from the one used in Graphical User Interface (GUI) testing [17, 18]. Unlike an EFG in GUI testing that only shows a simple transition relationship (i.e., the sequential flow of events in GUI), our proposed EFG represents a simple as well as complex set of event interdependencies (relationships) among events. The complex interdependencies among events are represented using connector nodes like "split" and "join" combined with "not," "and," "xor," and "or" operator nodes. An Event-Flow Graph for GUI testing represents events in a GUI component, whereas the proposed Event-Flow Graph depicts all possible event interdependencies among business events, state events, or the control events of a system. Moreover, user interaction with GUI captured through automation framework builds an Event-Flow Graph for GUI testing, whereas in the proposed approach, a model is build after analysis of events and the interdependency captured from event templates. Formally, an Event-Flow Graph is defined as:

An Event-Flow Graph (EFG) is a four tuple EFG = {V, C, T, F} where

- V = {v1, v2, ..., vn} is a finite set of vertices representing events in a system with an infinite looping condition where there is no start or exit or it may include two events designated as a Start (S) node and an Exit (E) node to indicate a system startup and exit events;
- C = {c1, c2, ... , cn } is a finite set of vertices representing connector nodes between events. These nodes consists of **"not," "xor-sp," "or-sp," "xor-j," "or-j," "fork," and "join**."
- T = {t1, t2, ... , tk }is a finite set of event transitions;
- $F \subseteq (V \times T) \cup (C \times T) \cup (T \times V)$ is the event-flow relation (i.e., a set of directed arcs involving event nodes, connector nodes, and transitions).

3.4 Possible causes of faults in an Event-Flow Graph

This section addresses the issue of finding possible causes of faults in an event-based system due to complex relationships and dependencies among events. What can happen due to faults in a system is that a system or application is dependent and cannot be generalized. However, one can generalize the possible causes of faults. A Fault Model is defined by using the EFG to show

possible causes of faults due to complex relationships and dependencies among events. Using this fault model, one can easily identify all of the possible faults that can occur in a given test scenario or a test case.

Fault Model

A Fault Model describes certain categories of faults that a test strategy must detect [34]. A Fault Model defines a small set of possible fault types that can occur in a system. Given a Fault Model, the goal is to generate a test set of T from an Event-Flow Graph in such a way that any fault in a system of a type described in a Fault Model is guaranteed to be revealed when tested against T. The proposed test scenario generation scheme is based on the following Fault Model that covers eight different types of faults:

1. **Event-XOR-Split Fault:** This fault occurs at an event-xor- split node. Consider the following case given in Fig. 3(c). E1 is a causative event of E2, E3, and E4 and E2, E3, and E4 are the triggered events of E1 with a xor-split dependency. In an ideal state, E1 should trigger exactly one of the events in a Trigger Vector (i.e., E2, E3, and E4). In a case when E1 has occurred but none of the events in a Trigger Vector gets triggered or more than one event gets triggered, an Event-XOR-Split Fault occurs.
2. **Event-OR-Split Fault:** This fault occurs at an event or-split node. Consider the following case given in Fig. 3 (d). E1 is a causative event of E2, E3, and E4 and E2, E3, and E4 are the triggered events of E1 with an or-split dependency. In an ideal state, E1 triggers one or more or all events in a Trigger Vector (i.e., E2, E3, and E4). In a case when E1 has occurred but none of the events in a Trigger Vector gets triggered, an Event-OR-Split Fault occurs. An Event-or split is more common in real time systems where a large number of independent events can be triggered together.
3. **Event-AND-Split Fault:** This fault occurs at an event-and- split node. Consider the case that is shown in Fig. 3(e). E1 is causative event of E2, E3, and E4 and E2, E3, and E4 are the triggered events of E1 with an and -split (fork) dependency. In an ideal state, E1 has to trigger all events in a Trigger Vector (i.e., E2, E3, and E4). A fault occurs if all of the events are not triggered.
4. **Event-NOT Fault:** This fault occurs in an event-not node. Consider the following case given in Fig. 3 (b). E1 is causative event of E2 and E2 is the triggered event of E1 with an event-not dependency. An Event-not connector node indicates the non-occurrence of an event. In this case a fault can occur if E1 occurs in the presence of an Event-not node. This is a special case of a Non-event event, as described by Ward and Mellor in [57].
5. **Event-AND-Join Fault:** This fault occurs in an event-and -join (merge) node. Consider the following case given in Fig. 3(f). E1 is triggered by events E2, E3, and E4 and E2, E3, and E4 are the causative events of E1 with a join (and-join) dependency. The causative events of E2, E3, and E4 occur and they trigger E1. A fault occurs if either: (a) E1 is not triggered even though E2, E3, and E4 have occurred or (b) E1 occurs even though all the causative events have not taken place.
6. **Event-XOR-Join Fault:** This fault occurs at an event-xor- join node. Consider the following case given in Fig. 3(h). E1 is triggered by events E2, E3, and E4 and E2, E3, and E4 are the causative events of E1 with an event-xor join dependency. In an ideal state, exactly one of the causative events E2, E3, or E4 should trigger E1. A fault occurs if more than one

causative event occurs to trigger E1.

7. **Event-OR-Join Fault:** This fault occurs at an event-or- join node. Consider the following case given in Fig. 3 (g). E1 is triggered by events E2, E3, and E4 and E2, E3, and E4 are the causative events of E1 with an event-or join dependency. In an ideal state, one or more or all of the causative events of E2, E3, or E4 should trigger E1. A fault occurs if none of the causative events occur but E1 is triggered.
8. **Synchronization Fault:** This fault occurs when some events take place before the completion of all preceding events. The reason for this fault is that events are not executed in a timely manner as per the precedence of relationships between them. This type of synchronization fault can be seen with a sequential flow of one event following another. It can also be seen in concurrent events, in which all parallel events are not triggered together at the same time.

3.5 Generate Test Scenarios from an Event-Flow Graph

This section addresses the issue of designing an algorithm to generate test scenarios that not only give appropriate coverage of the model but also detects the possible faults proposed in the fault model. In any model-based testing approach, the test coverage criteria and the technique for the automatic generation of test scenarios are two main aspects. So an algorithm is designed by using a combination of classical breath-first and depth-first searches, which generate test scenarios from an EFG. Our algorithm is also capable of detecting faults like synchronization faults, loop faults, and events occurrence faults. Such faults are not addressed in the existing event based test case generation approaches [17-20].

A “test scenario” is a set of test cases or test scripts along with the sequence in which they are to be executed. Test scenarios are test cases that ensure that all event paths are tested from start to end. Every test scenario in the proposed approach is an independent test that consists of a sequence of events that follow each other where each event is dependent on the occurrence of the previous event. Test scenarios are designed to represent both typical and unusual situations that may occur in an application.

Event Paths and Event-Flow Precedence Relations

In order to define test coverage criteria, possible event paths are to be defined that are possible to be found in an Event-Flow Graph. Precedence relations among events in an Event-flow Graph are used as the basis for defining such paths. Events in an Event-Flow Graph satisfy an identical set of partial order relationships. There are various precedence relations among events that are possible to occur in an Event-Flow Graph. Precedence relations that are denoted as ‘$\prec$’ over a set of events (SE) in an Event-Flow Graph are defined as follows:

1. If an event $E_i \in SE$ precedes another event $E_j \in SE$ in an Event-Flow Graph, then, there exists a partial order relation between two events E_i and E_j , which is denoted as $E_i \prec E_j$. It signifies that event E_i occurs before event E_j in a system.
2. If an event $E_i \in SE$ precedes a fork, a xor-split or an or-split and another event $E_j \in SE$ is the first event that exists in any thread originated from a fork, an xor-split, or an or-split, and then $E_i \prec E_j$.
3. If an event $E_j \in SE$ follows a join, a xor-join, or an or-join next and another event $E_k \in$

SE is the last event in any thread joining a join, an xor-join, or an or-join, then $E_k \prec E_j$.

4. If an event $E_i \in$ SE and another event $E_j \in$ SE are two consecutive concurrent events in a thread originated from a fork, an xor-split or an or-split or any thread joining an or-join, xor-join, or join where E_i exists before E_j, then $E_i \prec E_j$.
5. If an event $E_i \in$ SE and another event $E_j \in$ SE in an Event-Flow Graph are involved in a loop (cyclic relation), then, there exists two partial order relations between events E_i and E_j, which are denoted as $E_i \prec E_j$ and $E_j \prec E_i$. It signifies that both E_i and E_j are involved in a cyclic loop. These precedence relations are used to define various event paths for defining test coverage criteria.

Based on the above event precedence relations among events, various types of event paths are possible in an Event-Flow Graph, which are namely a non-concurrent path, simple basic path, cyclic basic path, and concurrent path. Each type is described in detail in Fig. 4.

a) A non-concurrent event path is a sequence of non-concurrent events (that is, events that are not executed in parallel) from the start event to an end event in an Event-Flow Graph, where each event in the sequence has at most one occurrence, except those events that exist within a loop. Each event in a loop may have at most two occurrences in a sequence. Also, all events satisfy the precedence relations among them. A simple basic path and a cyclic basic path do not involve parallel events so they are two non-concurrent event paths.

- Simple basic event path: A simple basic path is a sequence of events where each event in a path occurs exactly once and all events satisfy the precedence relations among them. In an Event-Flow Graph shown in Fig. 4, s→31→32→e is a simple basic path.
- Cyclic basic event path: A cyclic basic path is a like a simple basic path but it involves a loop. All events in a cyclic basic path also satisfy the precedence relations among them. In an Event-Flow Graph shown in Fig. 4, s→29→30→29→30→e is a cyclic basic path. In a cyclic basic path some of the events may occur more than once. In the above example, event 29 and event 30 occur more than once.
- Partial basic event path: A partial basic path is a path that has a sequence of events starting from a start event but terminates during another event instead of an end event. Each event in that path occurs exactly once and all events satisfy the precedence relations among them. In an Event-Flow Graph shown in Fig. 4, s→3→11→xor→12 is a partial basic path since it starts with s event node and ends on the event with the event ID 12.

b) A concurrent event path has concurrent events. It is a special case of event-flow graphs, which has both non-concurrent and concurrent events that satisfy the precedence relations

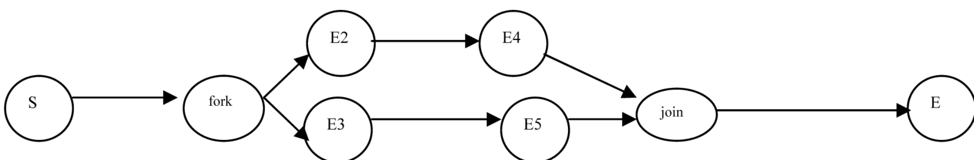


Fig. 4. Event-Flow Graph with concurrent events

among them. Concurrency among events occurs due to the presence of a fork, or-sp, xor-sp, join, or-join, and xor-join connector nodes in an Event-Flow Graph. For a complex and large system, it is common to have an explosion of concurrent event paths because there would be a large number of threads and on average every thread would have large number of events. In Fig. 4, an Event-Flow Graph with concurrent events E2, E3, E4, and E5 is shown.

In the Event-Flow Graph in Fig. 11, there are six precedence relations among events: S $\prec$ E2, S $\prec$ E3, E2 $\prec$ E4, E3 $\prec$ E5, E4 $\prec$ E, E5 $\prec$ E. There are six concurrent event paths that satisfy all of these relations that were specified above and consist of the same set of events.

P1 = s→ E2 →E3→E4→E5→ E
P2 = s→ E2 →E3→E5→E4→ E
P3 = s→ E2 →E4→E3→E5→ E
P4 = s→ E3 →E2→E4→E5→ E
P5 = s→ E3 →E2→E5→E4→ E
P6 = s→ E3 →E5→E2→E4→ E

Depending on the runtime environmental condition, the execution thread of a system would follow one of the concurrent event paths, but as to which concurrent event path would be followed cannot be known at an analysis level before the execution of a system. For effective testing with limited resources and time, the proposed approach aims to only test the relative sequence of concurrent and non-concurrent events (i.e., a set of precedence relations existing among these events). For this, one representative concurrent event path from a set of concurrent event paths is to be chosen that has the same set of events and satisfies the same set of precedence relations.

Test coverage criteria

Test coverage criteria [32] is a set of rules that guide in deciding on the appropriate elements to be covered to make test scenarios and test cases adequate. In an Event-Flow Graph, there are many, possibly infinite, paths between a start event and an end event. Structural test strategies, also known as "path-oriented" strategies, select a subset of paths to be executed. The selection of paths is aimed at covering a particular set of events in an Event-Flow Graph, along with the faults presented in the proposed Fault Model. The following two coverage criterion are defined for our proposed approach:

Event Coverage: Event coverage requires that each event in an Event-Flow Graph must be executed at least once. Such a requirement is necessary to check whether each event executes as expected. This kind of coverage criterion is able to detect synchronization faults but fails to cover other faults due to the fact that an event may occur independently from some other trigger point rather than through the connector node. Hence the execution of events through connector nodes remains uncovered (i.e., in Fig. 4, event 15 is triggered independently by event 13, and also through a xor and a fork connector node via two other paths). Thus, event coverage alone will not be able to detect faults due to an xor-split and a fork node (and-split). Due to these limitations, the proposed approach uses the Event Path Coverage criterion, as an improved test cov-

erage criterion.

Event Path Coverage Criterion: Event path coverage criterion is based on event paths that are identified in an Event-Flow Graph. This coverage criterion is used for testing both the loop (cycle) and concurrency among events in an Event-Flow Graph. In the proposed Event-Flow Graph events are interconnected using one of the special connector nodes (fork, join, xor-sp, or-sp, xor-j, or-j). Each transition emerging from an event node indicates a trigger for the next event. Testing branches in an Event-Flow Graph require that every branch leading to an event be exercised at least once under some test. Our aim is to ensure that Event-AND split, Event-OR split, Event-XOR split, Event-AND join, Event-OR join, and Event-XOR join faults are covered. Both the concurrent and non-concurrent event paths are used to cover these faults.

Algorithm for generating test scenarios

An algorithm called *GenerateEventPaths* has been proposed for generating event paths. In this algorithm the combination of both Depth First Search (DFS) and Breadth First Search (BFS) techniques are used. An Event-Flow Graph is traversed by the Depth First Search technique. except for a portion of the graph (which contains a fork node that initiates a set of concurrent events) with a type of node marked "fork." This then follows which sub tree is traversed following a Breadth First Search traversal of the graph. As discussed before, a Breadth First Search traversal helps to avoid a path explosion problem. It ensures that only one representative path is chosen from a set of concurrent event paths. Such a combination of two strategies is chosen to ensure that a path explosion problem is avoided due to the presence of concurrent paths. These event paths are not necessarily linearly independent paths due to multiple occurrence of a loop/cycle.

3.6 Generate Test Cases by augmenting test scenarios with necessary test information

This section addresses the issue of annotating information in test scenarios so that test cases are also generated in an event-based system. Whenever an event occurs in a system its occurrence has to be detected at an analysis stage. The condition or detection criterion becomes an annotation for an Event-Flow Graph (i.e., in an event where there is a "Guest request for a room" detected by the arrival of a "Booking request" in a system). Annotation is labeled with a string that describes test criteria in order to detect the occurrence of an event. Information about each event node in an Event-Flow Graph and its related event detection string is stored separately in a data structure, which is called the Event node Description Table (EnDT). This data structure is devised to facilitate the generation of test cases for each test scenario. At this level, details about each event such as the actions encapsulated in an event and the input and output parameters of each action are not considered in order to preserve the simplicity of an Event-Flow Graph. This information is used at the level of testing individual events in a system, which is not in the scope of this paper.

A test case in the proposed approach consists of the following components: an Event-Sequence Based Test Scenario and a sequence of transitions to detect an occurrence of events in a test scenario. Event-Sequence Based Test Scenarios constitute the expected system behavior. On the other hand, the sequence of transitions to detect the occurrence of events in a test sce-

nario makes up the source of the test input. The values for the sequence of transitions to detect events may be identified with the help of a system analyst. As a part of test case generation, the necessary values of all of the components of a test case are obtained from the corresponding Event-Flow Graph and its corresponding EnDT.

4. Case Study: Evaluating the Proposed Approach for a Real Time Application

We now present a case study for determining the effectiveness of the proposed approach described in the previous section. The following questions need to be answered to determine the effectiveness and usefulness of the proposed approach:

(a) Does the approach work for a wide variety of domain requirements?
(b) What is the level of accuracy and correctness in extracting events from the requirement specifications?
(c) How many different complex connector nodes can an Event-Flow Model handle?
(d) What are the variations in the length of test scenarios generated?
(e) How many and what types of possible faults can be detected in the test scenarios?

To answer the above questions while minimizing threats to external validity, a study was conducted using 11 different requirement specifications from the safety critical systems, real time systems, and event based systems. A detailed step analysis and the results from a case study, "Automatic Production Environment (APE)," on a real time system taken from the Real-Time Lab of Embry-Riddle Aeronautical University [27] is explained in detail in this section. Results of 10 other cases are discussed in the next section.

4.1 Generate Event-Flow Graphs

In order to generate an Event-Flow Graph, all events from the requirement specifications are extracted automatically. For this, requirements are first given to Stanford‘s Part of Speech (POS) tagger and then events are extracted from requirements by applying 17 parsing rules implemented in [58] on the output of the POS tagger to extract the list of SVO patterns (events) from the textual requirements in an XML format. All events extracted are automatically given an event ID and are stored in the XML file, as shown in Fig. 2. Other than that, events can also be found either in the manufacturer's system specifications (i.e., a functional description) or in the help facilities and handbooks. These sources also describe interdependency among events as well as desirable events in terms of system functions (responsibilities). Finding events at the requirement level from these sources is much easier than building other high-level design models. After applying the proposed steps of an event based OOA, as described in [1-3] for the APE case study [27], 34 events are extracted. These events are listed in the Table 1. Once events are extracted, they are documented by using the Event Templates. Event Templates are generated for each of the 34 events. A sample Event Template of one event from the events shown in Table 1 is presented in Table 2.

The event-flow diagram generator module automatically identifies causes and triggers relationships for every event by using its event template and it creates nodes as event identifiers and

edges as event-flow based on possible event interdependency, as discussed in Section 3.2. This construction process is fully automated. The resultant Event-Flow Model of the APE case [27]

Table 1. Events from the APE Case Study

1. User places package on the start place of Belt 1 (External Event).
2. User places package on the scanner part of Belt 1 (External Event).
3. User places package on the transition place of Belt 2 (External Event).
4. User places package on the end place of Belt 2 (External Event).
5. Sensor 1 senses the package at the start place (State/Control Event)
6. Sensor 1 generates a no-detect signal at the start place
7. Sensor 1 generates a detect signal at the start place
8. Sensor 2 senses the package at the scanning place
9. Sensor 2 generates a no-detect signal at the scanning place
10. Sensor 2 generates a detect signal at the scanning place
11. Sensor 3 senses the package at the transition place
12. Sensor 3 generates a no-detect signal at the transition place
13. Sensor 3 generates a detect signal at the transition place
14. Sensor 4 senses the package at the end place
15. Sensor 4 generates a no-detect signal at the end place
16. Sensor 4 generates a detect signal at the end place
17. Motor 1 starts conveyor belt 1
18. Motor 1 stops conveyor belt 1
19. Motor 2 starts conveyor belt 2 to move the package.
20. Motor 2 stops conveyor belt 2.
21. Motor 3 moves the scanner to the down position and waits for 10 sec.
22. Motor 3 moves the scanner to the home position and waits for 5 sec
23. Move the scanner to the down position and the limit sensor detects the down position.
24. Move the scanner to the up position and the limit sensor detects the home position.
25. Motor 4 moves the pusher to the extend position.
26. Motor 4 returns the pusher to the home position.
27. Pushing it forward and the limit sensor detects extended position.
28. Pushing it back and the limit sensor detects the home position.
29. System reports the status of the system (sensors and motors) after every two seconds to the web server.
30. User refreshes the status earlier than 2 seconds.
31. Package is removed from sensor 4 at the end place.
32. User toggles (manual / automatic) the control mode of the ALCS.
33. ALCS toggles (manual / automatic) the control mode.
34. ALCS reports the items track after every 5 sec.

Table 2. Events from the APE Case Study

1	Event ID	EA05	
2.	Event Name(verb phrase)	Sense package at the start place	
3.	Description	Sensor 1 sense the package at the start place (State/Control Event)	
4.	Initiator	Sensor 1	Count
5.	Facilitator	ALCS / Belt 1(Start place)	Count
6.	Affecter	Package	Count
7.	Timestamp		
8.	Causative Events (Preconditions)	EA01	
9.	Inputs		
10.	Trigger Vector	Sensor 1 generates a no-detect signal at the start place Sensor 1 generates a detect signal at the start place	
11.	Change-event	Connection between Sensor 1 and the package	

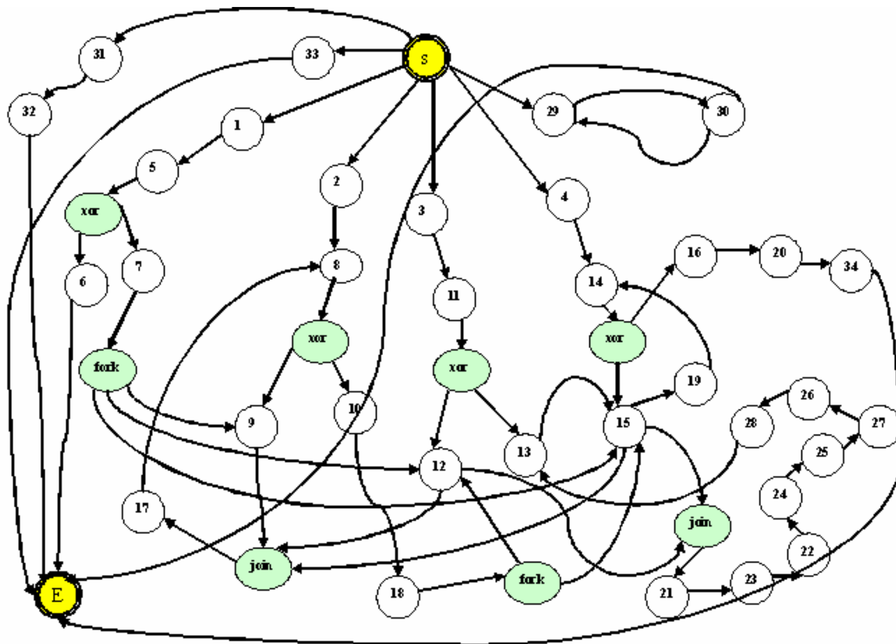


Fig. 5. Event-Flow Graph from the APE Case Study

after transformation into an Event-Flow Graph is shown in Fig. 5.

4.2 Generate test scenarios from Event-Flow Graphs

Using the *GenerateEventPaths* algorithm, event paths are generated from the Event-Flow Graph. These test scenarios have two different kinds of paths-partial paths and complete paths. Partial paths that end with nodes that lead to join nodes are left as such. These paths are not completed in our algorithm as they are partially able to detect the faults due to an Event-OR split and an Event-XOR split fault in the Fault Model. After applying the first part of our algorithm Generate_Event_Path to the Event-Flow Graph shown in Fig. 4, the following event paths are obtained:

1) s—29—30—29—30—E
2) s—4—14—xor4—16—20—34—E
3) xor4—15—19—14
4) s—3—11—xor3—12—
5) —xor3—13—15—
6) s—2—8—xor2—10—18—fork2—12—15—join2—21—23—22—24—25—27—26—28—13—
7) —xor2—9—
8) s—1—5—xor1—7—fork1—9—12—15—join1—17—8—
9) —xor1—6—E
10) s—33—E
11) s—31—32—E

In the generated paths, Paths 4, 5, and Path 7 are left as partial basic paths as they are partially able to detect faults due to an Event-XOR split fault in the Fault Model. Paths 1, 2, 10, and 11 are complete event paths in which 10 and 11 are simple basic event paths, whereas Path 1 is a cyclic basic event path. Paths 3, 4, 5, 7, and 9 are partial basic event paths. Path 6 and Path 8 are concurrent event paths. After applying steps from the second part of the Generate_Event_Paths algorithm, partially completed paths are converted to complete paths. Path 3 is completed by merging with Path 2. Path 6 is completed by merging with Path 5 and then with Path 3. Path 8 is completed by merging with Path 6. Final event paths are shown below:

1. s—29—30—29—E
2. s—4—14—xor4—16—20—34—E
3. s—4— xor4—15—19—14 —xor4—16—20—34—E
4. s—3—11—xor3—12—
5. s—3—11—xor3—13—15—
6. s—2—8—xor2—10—18—fork2—12—15—join2—21—23—22—24—25—27—26—28 —13—15—19—14 —xor4—16—20—34—E
7. s—2—8—xor2—9—
8. s—1—5—xor1—7—fork1—9—12—15—join1—17—8—
9. s—1—5—xor1—6—E
10. s—33—E
11. s—31—32—E

Path 4, 5, 7, and 8 are left as partial basic paths as they are partially able to detect faults due to Event-XOR split faults in the Fault Model.

The proposed algorithm to generate test scenarios selects a concurrent event path such that the sequence of all concurrent events encapsulated in that path corresponds to the Breadth First Search traversal of them in an Event-Flow Graph. This ensures that all precedence relations among events in an Event-Flow Graph are satisfied. One can avoid the generation of an entire set of concurrent event paths by finding the representative concurrent event path in an Event-Flow Graph. This will make the task of test scenarios generation process easier and at the same time a path explosion problem will be avoided, hence, reducing the testing efforts. (i.e., a path in Fig. 4, start —1—5—xor—7—fork—{9—12—15}-join—17—8—xor—10—18—fork—15—19—14—xor—16—20—34—E, is a concurrent path in which events 9, 12, and 15 are to be executed concurrently.)

4.3 Generate test cases based on test scenarios

For all 34 events of an Event-Flow Graph, as shown in Fig. 4, an Event node Description Table (EnDT) is shown in Table 3 that gives information about each event node and its related event detection string. The event detection string helps in finding when a particular event has occurred. Generated test cases obtained from the corresponding Event-Flow Graph in Fig. 4 and corresponding EnDT in Table 3 are shown in Table 4 along with all of the possible faults that can occur in a given test case based on the proposed fault model.

Table 3. Event node Description Table (EnDT)

Node ID	Event Name	Event Detection String
1	User places package on the start place of Belt 1	Mode=use; usercmd=place package; loc=start
2	User places package on the scanner part of Belt 1	Mode=use; usercmd=place package; loc=scanner
3	User places package at the transition place of Belt 2	Mode=use; usercmd=place package; loc=transition
4	User places package at the end place of Belt 2	Mode=use; usercmd=place package; loc=end
5	Sensor 1 senses the package at the start place	Generate S1signal=true
6	Sensor 1 generates a no-detect signal at the start place	S1signal=nodetect
7	Sensor 1 generates a detect signal at the start place	S1signal=detect
8	Sensor 2 senses the package at the scanning place	Generate S2signal=true
9	Sensor 2 generates a no-detect signal at the scanning place	S2signal=nodetect
10	Sensor 2 generates a detect signal at the scanning place	S2signal=detect
11	Sensor 3 senses the package at the transition place	Generate S3signal=true
12	Sensor 3 generates a no-detect signal at the transition place	S3signal=nodetect
13	Sensor 3 generates a detect signal at the transition place	S3signal=detect
14	Sensor 4 senses the package at the end place	Generate S4signal=true
15	Sensor 4 generate no-detect signal at end place	S4signal=nodetect
16	Sensor 4 generates a no-detect signal at the end place	S4signal=detect
17	Motor 1 starts conveyor belt 1	Motar1cmd=forward
18	Motor 1 stops conveyor belt 1	Motar1cmd=stop
19	Motor 2 starts conveyor belt 2 to move the package.	Motar2cmd=forward
20	Motor 2 stops conveyor belt 2.	Motar2cmd=stop
21	Motor 3 moves the scanner to the down position and waits for 10 sec	Scannercmd=scan
22	Motor 3 moves the scanner to the home position and waits for 5 sec	Scannercmd=home
23	Move the scanner to the down position and the limit sensor detects the down position.	Scannerdownsignal=true
24	Move the scanner to the up position and the limit sensor detects the home position.	Scannerhomesignal=true
25	Motor 4 moves the pusher to the extend position.	Pushercmd=extend
26	Motor 4 returns the pusher to the home position.	Pushercmd=home
27	Pushing it forward and the limit sensor detects extended position.	Pusherextendsignal=true
28	Pushing it back and the limit sensor detects the home position.	Pusherhomeosignal=true
29	System reports the status of the system (sensors and motors) after every two seconds to web the server.	Mode=auto;systemstatusmsg=true;repeat=true
30	User refreshes the status earlier than 2 seconds.	Mode=user;systemstatusmsg=true;repeat=true
31	ALCS toggles to the manual control mode.	Mode=auto;toggle=usercontrol
32	User toggles to the automatic control mode of the ALCS.	Mode=user;toggle=auto
33	ALCS reports the items track after every 2 seconds.	Mode=auto;autocontrolmsg= report item track;repeat=true
34	Package is removed from sensor 4 at the end place.	Mode=use; usercmd=remove pacakage; loc=end

Table 4. Test cases generated from an Event node Description Table (EnDT)

Test Case No.	Event Sequence	Sequence of Transitions	Faults Detected
1.	s—4—14—xor4—16—20—34—E	Mode=use; usercmd=place pacakage; loc=end; Generate S4signal=true; S4signal=detect; Motar2cmd=stop;Mode=use; usercmd=remove pacakage; loc=end	Event-XOR-split fault Synchronization fault
2.	s—4—14—xor4—15—19—14—xor4—16—20—34—E	Mode=use; usercmd=place pacakage; loc=end; Generate S4signal=true; S4signal=nodetect; Motar2cmd=forward; Generate S4signal=true; S4signal=detect; Motar2cmd=stop;Mode=use; usercmd=remove pacakage; loc=end	Event-XOR-split fault Synchronization fault
3.	s—29—30—29—E	Mode=auto;systemstatusmsg=true;repeat=true; Mode=user;systemstatusmsg=true;repeat=true; Mode=auto;systemstatusmsg=true;repeat=true;	Synchronization fault
4.	s—3—11—xor3—13—15—	Mode=use; usercmd=place pacakage; loc=transition; Generate S3signal=true; S3signal=detect; S4signal=nodetect	Event-XOR-split fault
5.	s—2—8—xor2—9—	Mode=use; usercmd=place pacakage; loc=scanner; Generate S2signal=true; S2signal=nodetect	Event-XOR-split fault
6.	s—1—5—xor1—7—fork1—9—12—15—join1—17—8—	Mode=use; usercmd=place pacakage; loc=start; Generate S1signal=true; S1signal=detect;S2signal=nodetect && S3signal=nodetect && S4signal=nodetect; Motar1cmd=forward; Generate S2signal=true	Event-XOR-split fault Event-AND-split fault Event-AND-join fault
7.	s—2—8—xor2—10—18—fork2—12—15—join2—21—23—22—24—25—27—26—28—13—19—14—xor4—16—20—34—E	Mode=use; usercmd=place pacakage; loc=scanner; Generate S2signal=true; S2signal=detect; Motar1cmd=stop; S3signal=nodetect && S4signal=nodetect; Scannercmd=scan; Scannerdownsignal=true; Scannercmd=home; Scannerhomesignal=true; Pushercmd=extend; Pusherextendsignal=true; Pushercmd=home; Pusherhomeosignal=true; S3signal=detect; Motar2cmd=forward; Generate S4signal=true; S4signal=detect; Motar2cmd=stop;Mode=use; usercmd=remove pacakage; loc=end	Event-XOR-split fault Event-AND-split fault Event-AND-join fault
8.	s—1—5—xor1—6—E	Mode=use; usercmd=place pacakage; loc=start; Generate S1signal=true; S1signal=nodetect	Event-XOR-split fault
9.	s—33—E	Mode=auto;autocontrolmsg= report item track;repeat=true	Synchronization fault
10.	s—31—32—E	Mode=auto;toggle=usercontrol; Mode=user;toggle=auto	Synchronization fault
11.	s—3—11—xor3—12—	Mode=use; usercmd=place pacakage; loc=transition; Generate S3signal=true; S3signal=nodetect	Event-XOR-split fault

5. Tool Support and Discussion of the Results

The Event-Based Test Scenario Generator Tool was developed to provide an integrated event-based one-stop solution for testing event-oriented systems. The prototype tool automates the entire process of generating an Event-Flow Graph from elementary events and computes test scenarios and test cases using the proposed algorithm. This tool was developed by using C# in Microsoft Visual Studio 2010 and .Net Framework 3.5, and it uses Graphviz (short for Graph Visualization Software), which is a package of open source tools initiated by AT&T Research

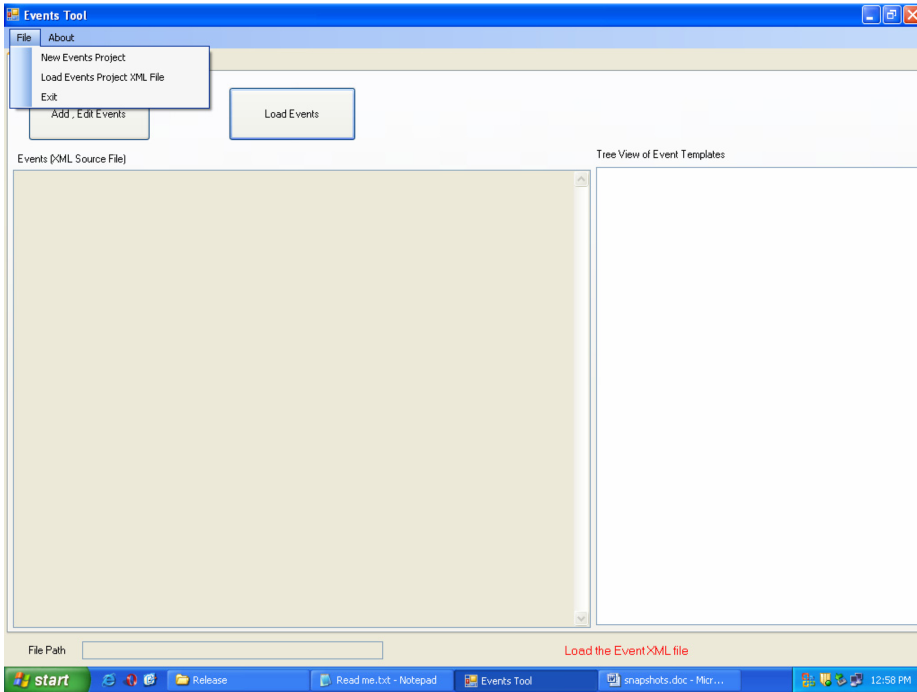


Fig. 6. Loading an XML file of event templates

Labs for drawing graphs specified in DOT language scripts. The tool also uses QuickGraph, which is a library containing generic graph data structures and algorithms. The tool enables a user to load an external XML file of event templates and then generates an Event-Flow Graph of events. It calculates all possible paths between the start and end node and gives an option to zoom and rotate the graph for easy viewing as well as save the generated graph as an image. Fig. 6 shows a snapshot where a user first loads an XML file containing all event templates and then on expansion all events are loaded from a file in a tree-like structure. As shown in Fig. 7, clicking on the “Compute Paths” or “Graph Image” button, calls a method to compute paths and graphing images. This method creates a dynamic Event Flow Graph in the system’s memory by using the “Quickgraph” Library. Vertices and edges are added to the graph depending on the type of dependency between events. Once a graph is created, paths are then calculated between the start and the end node of a graph by using the proposed algorithm in such a way that all of the nodes of a graph are covered at least once through the generated test scenarios. These paths are then displayed. Also a graph that is made in the memory of a computer is converted to a “graph.dot” file and this “graph.dot” file is displayed using Graphviz. This method also converts a “graph.dot” file into a jpeg image that can be opened later on to display the graph.

Using 11 different requirement specifications tests the effectiveness, efficiency, and scalability of the tool. The detailed steps and results of one of the graph are already presented in the previous section. Due to space constraints it is not possible to show all the cases so this section presents consolidated results for the remaining 10 requirements specifications in Table 5. Results are shown in terms of the 5 parameters described below:

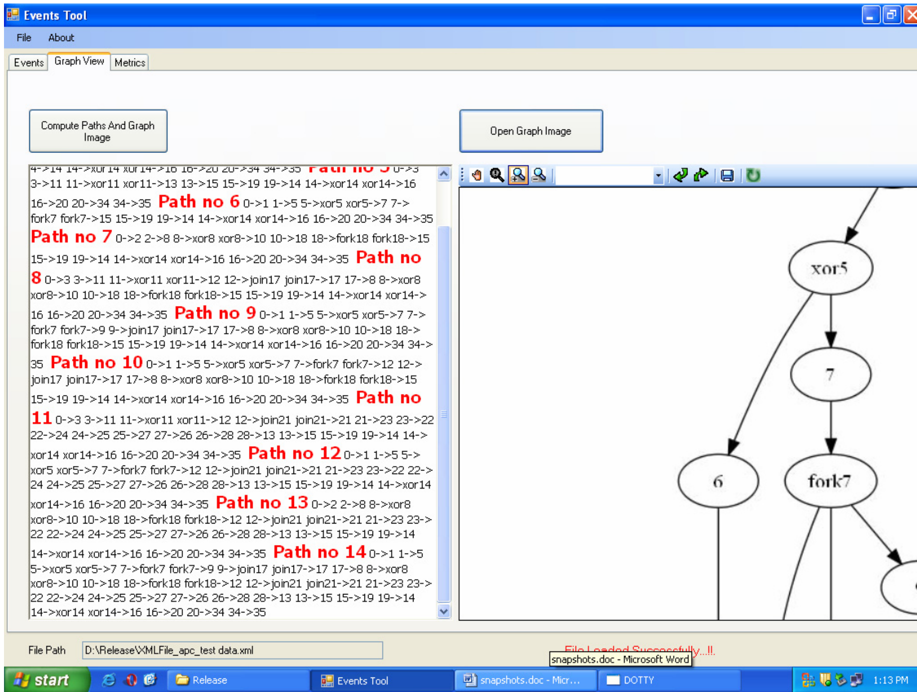


Fig. 7. Compute paths and an image of an Event-Flow Graph

Table 5. Evaluation results

S. No.	Related domain	NE_E	NE_T	T_{CN}	N_{TS}	N_{FD}
1.	Luggage Control Software	29	26	6	10	5
2.	Blood Bank	31	30	7	14	6
3.	Public Library Information System	43	39	4	18	3
4.	Avionics System	40	39	10	15	8
5.	Air Port Management System	20	12	12	15	9
6.	Automatic Production Environment	34	30	8	11	5
7.	University Management System	30	20	6	12	7
8.	Car Race Simulator	25	20	7	14	6
9.	Fighter Plane Control System	30	26	13	15	7
10.	Highway Intersection System	35	27	14	18	6

- N_{EE}- Number of events extracted by an expert
- N_{ET}- Number of events extracted by an event extractor module
- T_{CN}- Total number of event connector nodes (“not,” “xor-sp,” “or-sp,” “xor-j,” “or-j,” “fork,” and “join”)
- N_{TS}-Number of Test Scenarios generated by the Test Scenario Generator Module
- N_{FD}-Number of Faults detected in a test scenario

The effectiveness of the approach is evident from the results of the 11 case scenarios. In each case, the approach has been able to successfully extract events from the requirements, document

them, and also generate test scenarios and test cases. This has a significant impact on saving on the time and efforts required by analysts to do all these things manually. The scalability of the approach is evident from the fact that requirements were picked up from a wide range of applications of varying complexities from a simple Air Port Management System, Car Race Simulator, Luggage Control Software, University Management System, Fighter Plane Control System, Blood Bank, Automatic Production Environment, Highway Intersection System, Avionics System, all the way to a Public Library Information System. The approach and the tool are successfully able to handle events ranging from 20,25,29,30,30,31,34,35,40, to 43. These events were related with event operators ranging from 4,6,7,8,10,12,13, to 14. Test scenarios generated event paths of length ranging from 10,11,12,14,15, to 18. The faults related with "not," "xor-sp," "or-sp," "xor-j," "or-j," "fork," and "join" connector nodes were detected ranging from 3,5,6,7,8, to 9 in number.

6. Conclusion

Event based systems are rapidly gaining importance in many application domains ranging from real time monitoring systems in production, logistics and networking, to complex event processing in finance and security. This paper has presented a novel model-based testing approach to generate test scenarios and test cases using business events, state events, and control events that have been captured directly from requirement specifications. Several issues are identified in the beginning and contributions have been made for generating test cases and test scenarios using events from specifications. Various event interdependency operators are defined to represent relationships among events. These relationships are described as Event-Flow Interdependency. An Event-Flow Model is designed based on an identified Event-Flow Interdependency. This abstract model is stored as an Event-Flow Graph (EFG). A Fault Model is defined by using an EFG to show the possible causes of faults due to complex relationships and dependencies among events, whereas what can happen due to faults is system dependent and cannot be generalized. An algorithm was designed using a combination of classical breadth-first and depth-first searches, which generate test scenarios and test cases from the EFG. Our algorithm is also capable of detecting faults like synchronization faults, loop faults, and event occurrence faults. Such faults are not addressed in the existing event based test case generation approaches.

A process to annotate Event-Flow Graphs with the necessary test information has been described. This information is used at the level of testing individual events. A data structure, called an Event node Description Table (EnDT) has also been described. A prototype tool is developed to automate and evaluate the applicability of the entire process. Results have shown that the proposed approach and its supportive tools are able to successfully derive test scenarios and test cases from the requirement specifications of safety critical systems, real time systems, and event based systems.

References

[1] Singh, S.K., Sabharwal, S., Gupta, J.P. "*An event-based methodology to generate class diagrams and its empirical evaluation*", Journal of Computer Science, 6 (11), January, 2010, pp.1301-1325.

[2] Singh, K.,Sandeep, Sabharwal, Sangeeta, and Gupta, J.P., "*Event Patterns for Object Oriented Requirement Analysis*" in Proceedings of IASTED International Conferences on Advances in Computer

Science and Technology, April 2-4 , 2008, pp.115-120.

[3] Sandeep K. Singh, Sangeeta Sabharwal, J. P. Gupta: Events - An Alternative to Use Case as Starting Point in Object-Oriented Analysis. ICETET 2009: 1004-1010.

[4] McMenamin, Stephen M.; John F. Palmer (1984). Essential Systems Analysis. Prentice-Hall (Yourdon Press). ISBN 0132879050.

[5] Yourdon Edward (2003) "*Modern structured analysis*" Publisher Prentice-Hall India New Delhi 2003

[6] Monique Snoeck, Guido Dedene, Object-oriented modelling with events, TOOLS Europe 2000, Tutorial (3h), Tuturial notes and text available via http://merode.econ.kuleuven.ac.be/publications.aspx.

[7] Monique Snoeck, Geert Poels, "*Analogical Reuse of Structural and Behavioural Aspects of Event-Based Object-Oriented Domain Models*", Domaine Engineering Workshop, Proceedings of the 11th International Workshop on Database and Expert Systems Applications, London (Greenwich), 4-8 Sept. 2000,IEEE Computer Society, [PDF], pp.802-806.

[8] POELS G. "*On the Measurement of Event-Based Object-Oriented Conceptual Models*", 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, June 13 2000, Cannes, France.

[9] Bækgaard L. (2002), "*Event Modeling in UML*," Unified Modeling Language and Unified Process (part of IRMA'02 International Conference). Seattle, Washington.

[10] Bækgaard, L. (2004). Event-Based Activity Modeling. Conference on Action in Language, Action in Language, Organisations and Information Systems (ALOIS'04). Linköping, Sweden.

[11] Olivé, A,; Raventós, R. "*Modeling events as entities in object-oriented conceptual modeling languages*". Data&Knowledge Engineering 58 (2006), pp.243-262.

[12] J. F. M. Burg and R.P. van de Riet, COLOR-X: Linguistically-based Event Modeling: A General Approach to Dynamic Modeling, In Proceedings of the Seventh International Conference an Advanced Information System Engineering, LNCS (932), pp.26-39, Finland, 1995. Springer-Verlang.

[13] J. F. M. Burg and R.P. van de Riet, COLOR-X: Object Modeling profits from Linguistics, In Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing (KB&KS,95), Amsterdam, 1995, pp.204-214.

[14] Jog Roj & Martin Owen BPMN and Business Process Management - Popkin Software, September 2003.

[15] Stephen A. White - IBM and Derek Miers -Business Process Modeling and Reference Guide, BPM, Focus, September, 2008.

[16] Rittgen, P. Business Processes in UML, in Favre, Liliana: UML and the Unified Process, Hershey, PA: IRM Press, 2003, pp.315-331.

[17] Atif M. Memon, An Event-flow model of GUI-based applications for testing: Research Articles, Software Testing, Verification & Reliability, v.17 n.3, September, 2007, pp.137-157.

[18] Atif M. Memon , Mary Lou Soffa , Martha E. Pollack, Coverage criteria for GUI testing, Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, September 10-14, 2001, Vienna, Austria.

[19] Lu, Y., Yan, D., Nie, S., & Wang, C. (2008 December). "*Development of an Improved GUI Automation Test System Based on Event-flow Graph.*" Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Vol.02, 2, 712-715.

[20] Belli, F., Budnik, C.J., White, L.: Event-based Modeling, Analysis and Testing of User Interactions: Approach and Case Study. The Journal of Software Testing, Verification and Reliability, Vol.16(3) (2006), pp.3-32.

[21] Alessandra Russo , Rob Miller , Bashar Nuseibeh , Jeff Kramer, An Abductive Approach for Analysing Event-Based Requirements Specifications, Proceedings of the 18th International Conference on Logic Programming, July 29-August 01, 2002, pp.22-37.

[22] Alan S. Abrahams, David M. Eyers, Jean M. Bacon An Event-Based Paradigm for E-Commerce Application Specification and Execution. Proc. of 7th International Conference on Electronic Commerce Research, Dallas, Texas, 2004, 181-192.

[23] N. H. Gehani , H. V. Jagadish , O. Shmueli, Event specification in an active object-oriented database, Proceedings of the 1992 ACM SIGMOD international conference on Management of data, p.81-90,

June 02-05, 1992, San Diego, California, United States.

[24] Ping-peng Yuan, Gang Chen, Jin-xiang Dong, Wei-li-Han Reseach on an Event Specification for Event-Based Collaboration Support Software Architecture. Proc. Int Conf. On Computer Supported Cooperative Work in Design, NY, USA, 2002, 99-104.

[25] A. El-Ansary, "*Behavioral Pattern Analysis: Towards a New Representation of Systems Requirements Based on Actions and Events*," In Proc: SAC, 2002, pp.984-991.

[26] Requirements Definitions of Robotics Automation Using the Behavioral Patterns Analysis (PBA) Approach: The Production Cell System Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol-1 (CIMCA-IAWTIC'06), 2005, pp.883-892.

[27] http://www.rt.db.erau.edu/BLUE/02%20SE545_FA07_TEAM_BLUE_SRS_version2.pdf

[28] Allen, FE. (1970). Control flow analysis. Proceedings of a Symposium on Compiler Optimization. ACM Press: New York, 1970; 1-19.

[29] Rosen, B.K.(1979). Data flow analysis for procedural languages. Journal of the ACM 1979; 26(2): 322-344.

[30] R. V. Binder. Testing Object-Oriented Systems Models, Patterns, and Tools. Addison Wesley, Reading, Massachusetts, October, 1999.

[31] Ward, Paul T.; "*Stephen J. Mellor Structured Development for Real-Time Systems"*: Vol.2, Essential Modeling Techniques. Prentice-Hall (Yourdon Press). ISBN 0138547874. (ISBN 978-0138547875).

[32] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. ACM Computing Surveys, 29(4):366-427, December, 1997.

[33] T. H. Kim, I. S. Hwang, M. S. Jang, S. W. Kang, J. Y. Lee and S. B. Lee, Test Case Generation of a Protocol by a Fault Coverage Analysis. ICOIN-12., Tokyo, Japan, (1998).

[34] Aditya P. Mathur, Foundations of Software Testing, 1st Imp. : Pearson Education, 2008, 193-214, pp.45-46.

[35] Debasish Kundu, Debasis Samanta : "A Novel Approach to Generate Test Cases from UML Activity Diagrams", in Journal of Object Technology, Vol.8, No.3, May-June, 2009, pp.65-83.

[36] Chen Mingsong , Qiu Xiaokang , Li Xuandong, Automatic test case generation for UML activity diagrams, Proceedings of the 2006 international workshop on Automation of software test, May 23-23, 2006, Shanghai, China [doi>10.1145/1138929.1138931]

[37] H. Kim, S. Kang, J. Baik, and I. Ko. Test cases generation from uml activity diagrams. Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007, pp.556-561.

[38] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In *11th Asia-Pacific Software Engineering Conferen*ce (APSEC 2004), 2004, pp.284-291.

[39] Dong Xu , Huaizhong Li , Chiou Peng Lam, Using Adaptive Agents to Automatically Generate Test Scenarios from UML Activity Diagrams, Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), December 15-17, 2005 [doi>10.1109/APSEC.2005.110], pp.385-392.

[40] Chandler, R., Lam, C.P., Li, H.: "*AD2US:an automated approach to generating usage scenarios from UML Activity diagrams*." In: Proceedings of 12th Asia Pacific Software Engineering Conference, Taipei, Taiwan (2005) , pp.9-16.

[41] Huaizhong Li and C. Peng Lam : "*Using Anti-Ant-like Agents to Generate Test Threads from UML Diagrams*", TestCom 2005, LNCS 3502, 2005, pp.69-80.

[42] Chen, M., Mishra, P., Kalita, D.: Coverage-driven Automatic Test Generation for UML Activity Diagrams. In: ACM Great Lakes Symposium on VLSI (GLSVLSI) (May 2008).

[43] Philip Samuel, Rajib Mall : "*A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams*", e-Informatica Software Engineering Journal, Vol.2, Issue 1, 2008

[44] Dehla Sokenou "*Generating Test Sequences from UML Sequence Diagrams and State Diagrams*" IEEE Society October. 2006.

[45] M. Ebner. TTCN-3 Test Case Generation from Message Sequence Charts. In Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL), France, No-

vember, 2004.
[46] Li, Bao-Lin; Li, Zhi-shu; Qing, Li et al, Test case automate generation from UML sequence diagram and OCL expression, Computational Intelligence and Security, 2007 International Conference on 15-19 December, 2007 pp.1048-1052.
[47] Monalisa Sarma , Debasish Kundu , Rajib Mall, "*Automatic Test Case Generation from UML Sequence Diagram"*, Proceedings of the 15th International Conference on Advanced Computing and Communications, December, 18-21, 2007 [doi>10.1109/ADCOM.2007.49], pp.60-67.
[48] Emanuela G. Cartaxo, Francisco G. O. Neto and Patr´icia D. L. Machado Test Case Generation by means of UML Sequence Diagrams and Labeled Transition Systems 2007 IEEE International Conference on Systems Man and Cybernetics (2007) Publisher: Ieee, ISBN: 9781424409907 DOI: 10.1109/ICSMC.2007.4414060, pp.1292-1297.
[49] Lund, M.S., Stølen, K.: Deriving tests from UML 2.0 sequence diagrams with neg and assert. In: 1st International Workshop on Automation of Software Test (AST'06), ACM Press (2006) 22-28.
[50] Y.G. Kim, H.S. Hong, S.M. Cho, D.H. Bae, S.D. Cha : "*Test Cases Generation from UML State Diagrams*", IEE Proceedings - Software, Vol.146, No.4, August, 1999, pp.187-192.
[51] P. Samuel, R. Mall, A.K. Bothra : "*Automatic test case generation using unified modeling language (UML) state diagrams"*, IET Softw., Vol.2, No.2, 2008, pp.79-93.
[52] S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams. In SAICSIT '03, 2003, pp.296-300.
[53] Baikuntha Narayan Biswal , Pragyan Nanda ,Durga Prasad Mohapatra "*A Novel Approach for Scenario-Based Test Case Generation*" IEEE Society ICIT. 2008.
[54] S. Weißleder and D. Sokenou. Automatic Test Case Generation from UML Models and OCL Expressions. In Testing of Software - From Research To Practice (associated with Software Engineering 2008), (TESO'08), pp.423-426.
[55] M. Riebisch, I. Philippow, and M. Go¨ tze, "*UML-Based Statistical Test Case Generation*," Proc. Int'l Conf. Net.ObjectDays, Vol.2591, 2002, pp.394-411.
[56] Peter Frohlich and Johannes Link, "*Automated Test Case Generation from Dynamic Models*", ECOOP- Object Oriented Programming, Vol.1850, 2000, pp.472-491.
[57] Von Mayrhauser, A., France, R., Scheetz, M., and Dahlman, E. "*Generating Test-Cases from an Object-Oriented Model with an Artifical-Intelligence Planning System IEEE TRANSACTIONS ON RELIABILITY"*, Vol.49, No.1, March, 2000.
[58] Singh, K., Sandeep, Sabharwal, Sangeeta, and Gupta, J.P., "*E-XTRACT: A Tool for Extraction, Analysis and Classification of Events from Textual Requirements*", in Proc.of the 2009 International Conference on Advances in Recent Technologies in Communication and Computing, India, October, 2009, pp.306-308.

Dr. Sandeep Kumar Singh

Dr. Sandeep Kumar Singh is an Assistant Professor at JIIT in Noida, India. He has completed his Ph.D in (Computer Science and Engineering). He has around 11+ years' experience, which includes corporate training and teaching. His areas of interests are Software Engineering, Requirements Engineering, Software Testing, Web Application Testing, Databases, Internet and Web Technology, Object Oriented Modeling and Technology, Programming Languages, Distributed Computing, Model-based Testing, and Applications of Soft Computing in Software Testing. He is currently supervising 5 Ph.D's in Computer Science. He has around 15 papers to his credit in different international journals and conferences.

Professor Sangeeta Sabharwal

Prof. Sangeeta Sabharwal is currently working as Professor and Head of the Department for Information Technology at Netaji Subhas Institute of Technology in Delhi, India. Her areas of interest are Software Engineering, Metamodeling, Object Oriented Analysis, Software Testing, and Dataware House. Currently she is actively involved in applying different Soft Computing Techniques to different areas of software engineering, mainly in the area of testing. She has published papers in several international journals and conferences. She has also written a book on software engineering. Numbers of students are pursuing their Ph.Ds under her guidance. She can be contacted at: ssab63@gmail.com.

Professor J.P.Gupta

Prof. JP Gupta obtained a B.Sc (Engg.) from BHU in 1971, an M. Tech from the University of Roorkee (now IIT Roorkee) in 1973, and a Ph.D in Computer Engineering from the University of Westminster, London (UK) in 1985. Prof. Gupta has more than 37 years of teaching, research, and administrative experience. He has supervised 15 Ph.D theses and published more than 75 research papers in international journals and conferences. His research interests include Parallel Computing, Image Processing, Search Engines, and Software Engineering. He has been an IT consultant to a large number of organizations and completed several R&D and consultancy projects that were supported by government organizations and Indian industries.