Dealing with Concurrent Regions during Scenario Generation from Activity Diagrams

Robert Chandler, Chiou Peng Lam, and Huaizhong Li
Edith Cowan University
Bradford Street, Mount Lawley,
Western Australia 6050
(robert.chandler; p.lam; @ecu.edu.au – huaizhongli@ieee.org)

Abstract- ***Scenarios are a popular focus for the acquisition and validation of system requirements and in the generation of system-based test-cases. However, generating scenarios manually is a tedious process, which may introduce errors or produce incomplete scenario sets. This paper discusses an approach to scenario capture, in support of requirements engineering that can then be used for test-scenario capture and test-case generation. The approach has been automated successfully, producing usage-scenarios from UML (Unified Modelling Language) Activity Diagrams without the need for manual intervention. The paper walks through the approach - with a specific focus on the processing of concurrent regions - and compares the results with other approaches applied to the same UML activity model.***

I. INTRODUCTION

Scenarios are a helpful means of identifying and communicating system requirements [5], for verifying the completeness and correctness of a functional model [11], [23] and for aiding in the generation of test-cases [1], [17], [25] and [26]. Unfortunately, it is an extremely labour intensive process to capture them manually [24] and according to a recent study [22], there is a distinct lack of both methods and tools to support the seamless progression from modelling tool to Usage Scenarios (US), without the need for manual intervention.

Existing methods tend to use some form of manual derivation [3], [21], which may introduce errors or produce incomplete scenario sets [10]. Still other methods employ graphical scenario representations, such as Petri-nets [12], [15] or Message Sequence Charts [2], [14], [16] that do not form any part of the alleged de-facto standard for software development; the UML [4], [6], [9], [20].

This paper proposes an approach that uses the information exported from a UML model, in XMI (eXtensible Mark-up Language (XML) Metadata Interchange) format, to produce USs from Activity Diagrams (AD) that the model contains. This approach processes complex sequences of AD artefacts, that are regularly found within concurrent regions, which are often neglected in other usage-scenario generation approaches.

Activities that have looping or skipping paths make scenario capture difficult; this difficulty is compounded when looping or skipping occurs within a concurrent region; and more-so when in nested concurrent regions. Some approaches have also viewed the inclusion of multiple final and/or initial nodes in an AD as a difficult factor in the capture of scenarios [25], [26]; whereas this approach does not. The UML 2.0 specification [20] for AD modelling provides for the inclusion of multiple initial and final nodes at the basic-level of AD modelling. The approach discussed in this paper has been applied to models at both the Complete-Structured and the Intermediate levels for software development and process modelling respectively.

The rest of this paper is organized as follows. ADs are defined more formally in section II. The comparative approaches are introduced in section III along with some background on this area of research. We introduce the approach and algorithm in section IV and a description of the sequences of artefacts found when dealing with concurrency in ADs in section V. Section VII introduces the case-study/UML activity model upon which the approaches are applied and sub-section VII.A details the criteria used for comparing the processing results; which are presented in sub-section VII.B. Finally, our conclusions are drawn in section VIII.

II. AD DEFINITION

Typically, the modelling of an activity will begin with an initial node and progress through a set of edges and nodes until the activity ends at one or more final or flow-final nodes. We offer the following definition to formalize the structure of an AD:

$AD = \{V, E\}$

Let AD be an Activity Diagram within the set of ADs in a UML Model; furthermore, let:

V be a set of Artefacts where:

$A = \{a_1, a_2, \ldots a_n\}$ is a finite set of action elements;
$B = \{b_1, b_2, \ldots b_p\}$ is a finite set of branches;
$F = \{f_1, f_2, \ldots f_q\}$ is a finite set of forks;
$I = \{i_1, i_2, \ldots i_r\}$ is a finite set of initial nodes;
$J = \{j_1, j_2, \ldots j_s\}$ is a finite set of joins;
$M = \{m_1, m_2, \ldots m_u\}$ is a finite set of merges;
$O = \{o_1, o_2, \ldots o_v\}$ is a finite set of objects;
$Z = \{z_1, z_2, \ldots z_w\}$ is a finite set of finals or flow-final nodes;
$E = \{e_1, e_2, \ldots e_x\}$ be a finite set of edges.

Based on these sets of artefacts, an AD consists of sequences of vertices and edges that are connected according to the semantic rules established in the UML specifications.

$AD = \{i_n, e_1, (a_1 \vee b_1 \vee f_1 \vee m_1 \vee o_1 \vee z_1), e_2, (a_2 \vee b_2 \vee f_2 \vee j_2 \vee m_2 \vee o_2 \vee z_2), \ldots e_n, (a_n \vee b_n \vee f_n \vee j_n \vee m_n \vee o_n), e_{n+1}, z_n)$

To direct the flow of control and data throughout an activity, decision edges may carry guard conditions to determine which outgoing edge will receive 'control' to continue the activity. We define guard conditions thus:

T. Sobh (ed.), Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, 415–420.

$G = \{g_1, g_2, \ldots g_x\}$ is a finite set of guard conditions; and

G_x is in the corresponding edge e_x such that:

for all G_x *there exists* e_x

III. Related Works

The US Department of Defence (DoD) prepared a guidebook [18] defining a set of procedures to establish a structured and disciplined approach to integration testing from end-to-end (E2E). The E2E guidebook describes usage scenarios as thin-threads; stating that a thin-thread is a complete scenario from the end-user's point of view. A thin-thread describes just one operation from beginning to end, either successful or not.

Related thin-threads and test-scenarios can be grouped together to collectively describe all possible outcomes for an activity; for instance, (to use a well-worn example), when an ATM customer selects to withdraw money from a savings account there may be several outcomes. The obvious 'successful operation', where everything goes as planned with the customer withdrawing the desired amount from their own account; and then the other unsuccessful outcomes that may also occur; such as when our hypothetical customer either:

1. enters a wrong PIN number;
2. inserts the wrong card type;
3. selects the wrong account type; or
4. attempts to withdraw an amount larger than is available.

All of these scenarios, (and possibly others that are not included), make up an ATM *withdrawal* activity, or 'thin-thread group'. Each of these individual outcomes can be referred to as either a thin-thread or a US. For our purposes, we choose to refer to these outcomes as USs rather than thin-threads; a term we reserve for approaches focusing specifically on the development of test-cases.

[3] presents an approach to extracting thin-threads from Ads; and although their approach is quite exhaustive, it requires a considerable amount of manual processing in the conversion from an AD to an Activity Hyper-Graph and again in the collection of conditional expressions. The approach does not attempt to maintain concurrency and serializes the vertices discovered in Concurrent Regions (CR). There is currently no tool support for the approach.

[25] introduces a gray-box approach and tool (UMLTGF) for generating test-scenarios from ADs. Their approach, like that of [3] could not be fully automated. CRs are restricted to a maximum of two threads that have sequential 'CallActions' only in [25]. Interestingly, their algorithm does not appear to differentiate between branch nodes and fork nodes. Even with the 'two-thread restriction' we were unable to capture a pair of threads in a single scenario; thus, resulting in the capture of each thread as an alternate trace. This means that their approach may exclude many scenarios from a model, while at the same time producing incomplete traces.

[26] presents an approach that employs bacteria-like agents, optimizing an earlier algorithm [13] to extract thin-threads from ADs. This approach for generating test-scenarios directly from ADs is implemented in a tool called TSGAD (Test Scenarios Generator for Activity Diagrams). TSGAD lists the sets of test-cases derived from the artefacts between an AD's initial and final nodes in an output log file. These sets of test-cases can be clustered together to form a path through the AD artefacts. From these clustered paths it can be established which USs have been captured and which have not. They do not place the same restrictions on the structure of CRs and allow the inclusion of branches, merges and nested CRs.

IV. The Approach

Although traversing the nodes and edges in a directed graph such as an AD, is not generally considered a difficult task – several algorithms perform this duty effectively such as DFS[1] and BFS[2]– when iteration and recursion are involved the difficulty is compounded. To ensure that a scenario does not traverse edges unnecessarily, the proposed method limits the number of times an edge can be traversed by applying a *status* variable to each AD artefact. A vertex's status can either be ACTIVE or INACTIVE; while an edge's status can be either FINISHED or UNFINISHED. The default status for a vertex is INACTIVE and an edge's default status is UNFINISHED.

During the processing of an AD for USs, each edge's source vertex has its status tested. If the current edge's source vertex is ACTIVE, then that vertex has already been encountered during the capture of the current scenario; therefore, the current edge's status is reset to FINISHED and it is processed once more as usual. This ensures that, outside of nested iterative loops, each edge is traversed at most twice.

When an edge with a FINISHED status is encountered, that edge is ignored and the process then moves to the next edge; or it begins to recurse until it finds an alternative route. This ensures that a branching edge (even an action node's self iterating edge) does not result in an endless loop or produce a scenario that is already represented with fewer iterations of the same trace-route. Each looping guard condition is tested in both the true and false conditions only; without performing boundary value analysis whereby, each potential expression value is tested at inside and just outside the expected limits for the situation.

When nested looping constructs are encountered there is the potential for scenario explosion and therefore, the number of times that an edge is traversed must again be limited. The situation depends on the number of outgoing edges that a nested branch node contains. Each internal edge must be traversed at most twice for each external branch's outgoing edge.

Recursion is a natural method of traversing directed graphs, but the problem becomes somewhat difficult when processing the nodes and constructs within a concurrent region. Particularly when one or more of the concurrent threads contains a looping or skipping construct. These constructs are identified by the sequence of the branch-merge node pairs. A

[1] Depth-First Search
[2] Breadth-First Search