2005

AD2US: An Automated Approach to Generating Usage Scenarios from UML Activity Diagrams

Robert Chandler
Edith Cowan University

Chiou Peng Lam
Edith Cowan University

Huaizhong Li
Edith Cowan University

10.1109/APSEC.2005.25

AD2US: An Automated Approach to Generating Usage Scenarios from UML Activity Diagrams

Robert Chandler
Edith Cowan University
School of Computing
and Information Science
2 Bradford Street, Mt Lawley
Western Australia
robert.chandler@ecu.edu.au

Chiou Peng Lam
Edith Cowan University

Huaizhong Li
Edith Cowan University

Abstract

Although attention has been given to the use of UML (Unified Modelling Language) activity diagrams in the generation of scenarios, thin-threads and test-cases, the processes described in the literature rely heavily on manual intervention either in the information extraction process or in the process of transforming them to an alternate structure. This paper introduces an approach that will capture, store and output usage scenarios derived automatically from UML activity diagrams.

1. Introduction

In evaluating large, complex UML (Unified Modelling Language) analysis and design models, Berenbach[4] argues that the ultimate goal of requirements analysts is to be able to develop a complete set of requirements without complex tools or extremely specialized competencies. He states that some researchers have "adopted approaches that require mathematical skills beyond those that a business analyst might ordinarily be expected to possess"[4].

UML Activity Diagrams (AD)s are commonly used to model business processes, basic control and data flow in software systems and they require little technical expertise to develop and understand. A Cutter IT[9] review of an Armstrong article suggests that ADs can help reduce the incidence of the "blank page" syndrome encountered by some requirements analysts and modellers. ADs can be used as simple flow-charts or for detailed representations of behavioural logic[13]; making them versatile enough to be used by novices and professionals alike.

Barros[3] suggests that while reviewing the literature regarding the UML, it is easy to notice that when compared to the other UML diagram types, ADs have been given little attention. He suggests that this is possibly due to ADs association with state diagrams in earlier versions of the UML 1.x [3]. In the UML 2.0, ADs are finally separated from the state machine specification and are more closely associated with Petri-nets. Barros[3] predicts that this change along with the added activity node types and definitions, will increase the capabilities of AD modelling and he believes that this added modelling power will give rise to more frequent use of ADs[3].

Kösters[14] points out that of the behavioural diagrams which are administered by the UML, only the AD is capable of successfully specifying an entire set of use-case scenarios in a single diagram. In addition, ADs can help to validate the completeness and correctness of both textual and graphical use-cases[9]. They are potentially a rich source of test related information in both business and software-based models, which can be harvested early in the design phase. [9] points out that ADs can be used to identify candidate test cases that represent typical usage scenarios "such that a minimum number of test-cases provide the largest amount of requirements coverage"[9].

Regnell[20] states that an obvious motivation for combining scenario-based requirements engineering with verification and validation is "the opportunity to minimize modelling effort by using the same information for several purposes"[20]. They assume that combining scenario-based requirements engineering with verification and validation will promote traceability from requirements, through design to testing. They conclude that using scenarios for testing is an important area for further study; bearing in mind that this conclusion is made prior to the release of the UML 1.5 and long before UML 2.0. Hence, the importance of AD based research must now be paramount.

Deriving all the possible USs (Usage Scenarios) from

Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)
0-7695-2465-6/05 $20.00 © 2005 IEEE

each activity diagram in a UML model, is a very time consuming activity when performed manually. In addition, the literature indicates that deriving scenario-based information from ADs, for any purpose, is a complex process and many of the techniques described in the literature use some form of manual intervention [2, 16, 23]. This not only extends the time needed for the tasks being performed, but also increases the risk of introducing faults.

This paper presents an approach, dubbed *AD2US*, that automatically extracts USs from ADs; thereby extending the time available for other activities such as test-case generation or the verification of consistency between ADs, use-cases and usage scenarios. Currently, this study is concentrating on the collection and storage of data while maintaining the context of its source; then using information to generate a list of USs that can be viewed in tree and table format.

The rest of this paper is structured as follows; in section 2 a brief review of relevant works is presented and section 3 is a background to the area of study. Section 4 offers an outline of our solution and a discussion of the significance and benefits of this work. Finally, section 5 contains our conclusions and an introduction to possible future works extending this research.

2 Related Work

Usage scenarios are derived from software models for various purposes, including assisting in the transition from requirements to design[1, 12] as well as for requirements or design validation[20, 14] and for test case generation[2, 5, 15, 11].

2.1 From Requirements to Design

Amyot[1] describes an approach to transforming Use Case Maps to other scenario based definitions such as Message Sequence Charts using XSLT (XML Stylesheet Language Transformations). They suggest that their process will be useful for the early validation and synthesis of design models. Although, neither the graphical source or the target representations used in this research are related to the UML, which is rapidly becoming the pseudo modelling standard for the software development industry[21].

Jarke[12] gives us an insight into the important part that scenarios play in the development of good design models. Scenario usage-technique selection is one of the most crucial topics to be addressed according to Jarke and should be "based on sound cost-benefit analysis"[12][1].

[1] Page 48.

2.2 Design Validation

Regnell[20] discusses the need for combining scenario analysis with Validation and Verification. He suggests that a major reason scenarios are not more commonly used in testing, is that the scenarios generated during requirements analysis are "out of date by the time a system is ready for testing"[20]. Therefore, having an automated method of capturing usage scenarios could improve the quality of the final product. While also helping to update the status of the documented scenarios captured during early phases in the development life-cycle.

Kösters[14] introduces a method for the proper coupling of structural and behavioural aspects of a UML design again for validation and verification purposes, using *refined* ADs. ADs can depict control and data flow, Kösters suggests that coverage criteria for program testing be carried over to the validation of use case models. Furthermore, he suggests that the validation of an AD "should not stop before 100% of Vertex coverage has been achieved. Therefore, having an automated way of generating every possible scenario, for each use case, which provides adequate coverage for walk-through analysis must be worthwhile.

2.3 Test Case Generation

Briand[5] presents an approach to UML-based system testing using UML ADs to capture the sequence of usage-scenarios related to each of the use-cases (UCs) within a system model. The UC based AD is then manually transformed into a weighted graph, which makes the information more amenable to graph analysis. Once the sequence of USs is identified, they are able to determine which paths are active throughout each scenario's operation and should therefore undergo testing.

Tsai[22] offers an approach for End-to-End (E2E) Integration Testing, described originally by Paul [19], with user-oriented test scenarios for functional regression testing. The test scenarios are derived from textual scenario descriptions rather than from graphical representations such as UML UCs and ADs.

Bai [2] presents an approach that reduces UML ADs to a type of activity hypergraph, which was originally introduced by Eshuis and Weiringa [10]. Bai's algorithm requires a manual pre-process stage which is performed prior to the generation of a thin-thread tree, a condition tree and a data-object tree.

3 Background

Firstly, let us define UCs and scenarios. A **use-case** is a collection of possible scenarios representing a set of interactions between an actor and the subject system [7]. The

possible scenarios are representative of both successful and unsuccessful interactions, where the interaction goal is either achieved or fails. A **scenario** is a series of actions that occur under specific conditions that result in a particular outcome; again representative of either a successful or unsuccessful outcome with regard to the interaction goal of a particular UC.

One method of defining UCs is to textually describe them using a template to ensure that all possible scenarios and usage interactions are defined. Cockburn[6] provides various formats of a popular template on his website. Another method of defining UCs produces them as UML UC diagrams. The UC diagram depicts an actor/s interacting with a system and describes the purpose or goal of that interaction. Each UC typically has multiple scenarios as previously mentioned and there are a number of ways to depict them. Each individual scenario can be portrayed textually or graphically using an interaction diagram; such as a sequence diagram, communication diagram or an AD.

ADs can also represent *all* the possible outcomes or scenarios of a UC using the one diagram; making them useful for verifying the consistency, completeness and correctness of both textual and graphical UCs and scenarios alike[14].

UML ADs are developed using elements that are divided into two groups; Nodes[2] and Edges. The OMG's UML 2.0 superstructure specification [18][3] defines three types of Nodes; Action nodes, Object nodes and Control nodes; while Edges are defined as the transitions that represent control flow and data flow between nodes. The basic UML AD elements are depicted in figure 1.

Figure 1. A snapshot of activity diagram elements

The UML 2.0 describes the elements depicted in figure 1 in the following manner:

- An Action node is a 'fundamental unit of executable functionality in an Activity.' Actions have incoming and outgoing edges that signify control flow and/or data flow from and to other nodes respectively. A Compound Activity is a node in an AD, which is a condensed set of nodes and edges depicted in a separate diagram.
- An Object node is a node that indicates an instance of a particular classifier, that may be available at a particular point in the activity.
- A Control node is used to coordinate the flow of data and control between other nodes. 'Control nodes' include decision and merge nodes, fork and join nodes, initial nodes and final nodes[4].

The UML 2.0 superstructure[18][5] also describes several levels of activity modelling; Basic, Intermediate, Complete, Structured, Complete-Structured and Extra-Structured.

Figure 2 is an AD at the basic level of design, which represents a simplified ATM login and session creation activity.

The activity begins at the *Ready* initial node and progresses toward one of the final nodes depending on the evaluation of the conditions that are met during the activity. For instance, the first scenario can leave the initial node and progress through the *ATM Card* object to the *Get Card Details* action and then, depending on the condition set on this node's outgoing edge, end at the *Final_State_1* final node; so completing the activity. However, the next scenario can begin the same way, but instead of progressing toward *Final_State_1*, the system can progress from the *Get Card Details* action towards the *Get USER PIN* action and end at the *Final_State_2* final node.

The conditions that may be associated with edges in this type of diagram, use the factor format t[g]/e, where t = trigger, g = guard and e = effect; incidently, each of these factors is considered optional. In addition, OCL[17] (Object Constraint Language) constraints can be applied to activities in general, or to individual actions within an activity. In the next section, we introduce the AD2US process.

4 The AD2US Process

Firstly, we discuss the modelling and exporting related processes followed by the the capture and storage of the required data. This is then followed by a discussion of the the scenario generation process.

4.1 Modelling and Exporting

Poseidon 2.4.1 is the selected modelling tool for designing and exporting UML activity-diagram-based model; although this version has limitations relating to compliance with the UML 2.0 and activity modelling, it does provide the functionality to produce the level of diagrams desired. Furthermore, it provides efficient XMI-format exporting. The structure of the exported data produced by this version of Poseidon is common to other versions and tools; such as the community edition of Poseidon, ArgoUML and it is

[2] The OMG uses the term Node, whereas in general activity graph terms they are referred to as Vertices or an individual Vertex

[3] page 303.

[4] activity final and flow final

[5] page 265.

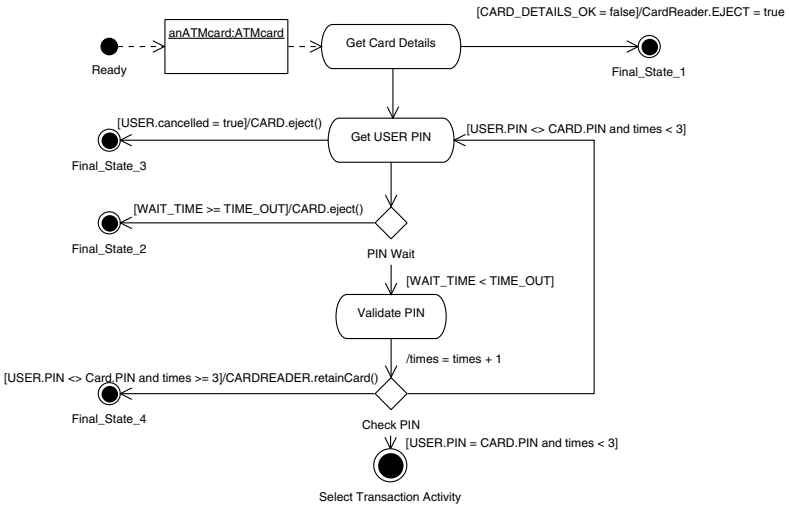

Figure 2. create a session activity diagram example

similar to the exported output produced by Together Control Center. This makes the focused nature of data capture reasonably generic, showing that the approach can be applied to other modelling tools.

4.2 Data Capture and Storage

Once a UML model has been exported and is ready for processing, the file is read into a DOM (Document Object Model) using the Java programming language. The reason DOM based processing is used instead of SAX based processing, is that the information within an XMI file is not necessarily sequential. For instance, AD information is spread over several regions in the document. An AD's identifier, name and graphical meta-data occupy a section that is quite early in the document.

This graphical section has an identifier attribute that directs processing to a much later section in the document containing the activity graph information. This is where the node and edge details are held. Toward the very end of the document is where any OCL-based constraints are located. The constraint objects contain an identifier that directs us back to the activity graph or the element that it constrains. The DOM parser allows us to move around the document to locate objects or elements that we are processing; whereas SAX-based parsing is a one-way process; hence, we would be unable to jump around the document to locate the objects for which we are searching.

As an AD is encountered in the DOM, AD2US captures the details of the diagram; such as its identifier and name, and then searches the XMI structure for the corresponding activity graph. When the appropriate activity graph is located, the details of each diagram element are extracted and deposited into a data structure that is associated with the specific diagram's identifiers.

The structure used for this storage is a kind of dynamic array called a Vector, which makes manipulation of the contents quite easy. This capture/storage process is applied to each AD found in the model resulting in a structure containing only information relating to diagram elements and their attributes and projected associates. Their associates include incoming and outgoing edges for all node types; and source, target, guard, trigger and effects for all edges. Along with this information, local constraints and activity-based constraints can also be captured and stored.

Although the information is already contained in an exported XMI file, tracing the relationships between the elements of an activity is difficult due to the size and structure of XMI files. In some cases, an XMI file can carry hundreds of thousands of elements, most of which are related to the model's graphical meta-data. With the XMI validation DTD being quite loose, modelling tools can also include proprietary information throughout an XMI file, making the task of locating information and the readability of the file's contents difficult.

Only a small percentage of the data contained in an XMI file describes the actual AD elements themselves; for example, the name, id, type of element and the associated incoming and outgoing identifiers or other edge details. In fact, one of the models designed for use in this research is of a simple Automated Teller Machine (ATM); this model contains eight UC diagrams, seventeen class diagrams, one sequence diagram, six state diagrams, two deployment diagrams and six structured-level ADs.

When this model was exported to XMI, the resultant file contained more than 159,000 elements. Once filtered for the information associated with the ADs, the total number of elements was reduced to 700. It was then determined that

we could reduce the processing time required for automating the filtering and manipulation of the data in an XMI file significantly, by capturing and storing this information separately. Therefore, the information is not only stored in a local Vector for manipulation throughout the process, but it is also stored in an XML file for use in external processes, such as US generation.

4.3 Usage Scenario Generation

AD2US applies a modified version of the DFS (Depth-First Search) algorithm[8]. This is an elementary graph processing algorithm which uses recursion to trace paths through the vertices and edges in either a directed or undirected graph; making it very compatible with XML/XMI processing. During the AD2US process described in section 4.2, as each element in the activity graph is encountered, the outgoing or target identifiers and/or the incoming or source identifiers are stored with the element.

Each element directs processing to the next element in the scenario, using the identifier associated with the current element's outward bound item/s. For instance, an edge must contain only a single source identifier and a single target element identifier; whereas various nodes can have both multiple incoming edges and/or multiple outgoing edges. Such as action and object nodes, which allow both multiple incoming and outgoing edges; while fork and decision nodes have a single incoming and multiple outgoing edges. Merge and join nodes on the other hand, have multiple incoming edges and a single outgoing edge. Obviously, initial nodes can have only one outgoing edge, while final nodes can have only one incoming edge. A simplified representation of the approach is depicted in figure 3.

The approach continues while there are unprocessed ADs in the XMI file. It traces the AD's associated Activity Graph and begins by locating the Initial node within each diagram. Next, we follow the outgoing edge's target ID to the next node. When a final node is encountered, the process outputs the scenario. Otherwise, the details of other nodes are stored and then each outgoing edge is followed. The algorithm relies upon the tagging of encountered elements, both vertices and edges, with one of three descriptors. These descriptors are: UNDISCOVERED; DISCOVERED and FINISHED. Each element is initialized with the UNDISCOVERED tag when it is added to the storage vector. Figure 4 depicts the modified algorithm in pseudo-code.

When the US capture process begins, as each element in the vector is encountered, an object of that element's type is instantiated and its status attribute is set to DISCOVERED (line 10). In the example, the figure then shows the element being output in some way at line 11. The algorithm then tests for the elements *type*; as edges must have only

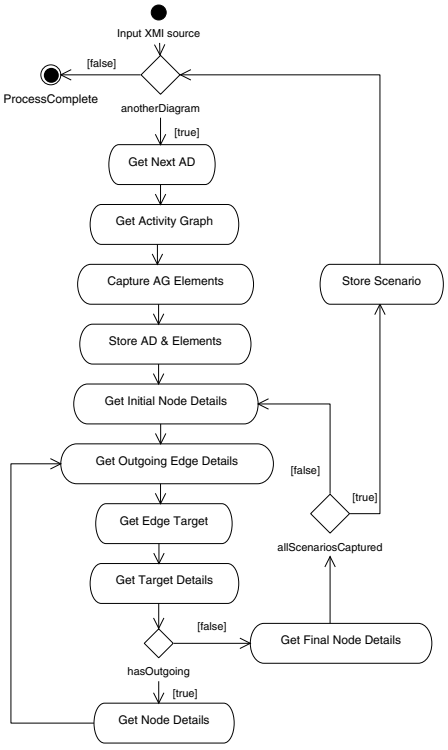

Figure 3. Simplified Approach AD

```
1.    findInitialNode()
2.      for each element in the Vector
3.        if element.type == "initial"
4.          element.status = DISCOVERED
5.          processThisElement(element.id)

6.    processThisElement(ID _id)
7.      for each element in the Vector
8.        if element.id == _id
9.          element.status = DISCOVERED
11.           output element.scenario  /*A string representation*/
12.             if element.type == "edge"
13.               processThisElement(element.target.id)
14.             else
15.               for each outgoing edge
16.                 processThisElement(element.outgoing.id)
17.         if element.status == DISCOVERED
18.           output element.scenario
19.             if element.type == "edge"
20.               processThisElement(element.target.id)
21.             else
22.               for each outgoing edge
23.                 processThisElement(element.outgoing.id)
24.               element.status = FINISHED
25.         if element.status == FINISHED
26.           /* do nothing */
```

Figure 4. modified depth-first search algorithm

one target identifier, this edge's target id is sent to be processed next. If on the other hand the element is not an edge, AD2US then processes any and all of this vertex's outgoing edges.

At lines 15, 16 and again at lines 22 and 23, AD2US calls the *processThisElement* procedure, passing it the id for each *outgoing* edge that belongs to the current node. That way, all the edges leading from each node in the activity are traversed and thereby given the opportunity to be discovered and finished. If however, the current element happens to be an edge, then the *processThisElement* procedure calls itself this time passing the *target* id of the current edge. Once the final node in a scenario is processed, the algorithm steps to line 24 setting its status attribute to FINISHED; and thereby locking that element from future processing iterations.

An issue that was encountered during scenario generation relates to a kind of scenario explosion. However, it does not relate directly to the kind of scenario explosion described by [7], where an analyst tries to list all possible interactions with a system. We use the term to describe the situation where an AD can include designed-in iteration or recursion. In these situations AD2US could fall into an infinite loop where the process cannot escape traversing an edge that returns to an earlier node in the scenario; such as that depicted in figure 2 after the *checkPIN* control node.

Without some way of breaking the loop the process reaches this edge and hence continues to iterate. Each cycle through this set of nodes and edges may be seen as part of a new scenario by AD2US. In the diagram, a guard condition ([USER.PIN $\ll\gg$ CARD.PIN and times $\ll$ 3]) is offered. The system being developed can use this to ensure that the edge is not traversed endlessly, but AD2US must break the cycle without understanding this expression. The problem we recognised is that not all guard conditions could be used to stop this scenario explosion.

Although part of the guard condition demands that **times $\ll$ 3**, for AD2US to use guard conditions, it would either need to replace variables with values like [22], or it would need to parse and recognise guard expressions; then it would need to create and instantiate variables according to their type, name and value ranges, as well as then increment/decrement them as required. A prospect that would significantly extend the scope and time available for this component of the study.

Instead, we determined that it is possible to capture all the scenarios by calculating the number of final nodes that are downstream of a node where there is a convergance of incoming paths such as that in the action node depicted in figure 2 as *Get USER PIN*. This method works fine in all situations where there is only one edge that directs the activity to an already processed node.

For brevity, a shortened sample of AD2US output is presented in figure 5; the model used in the example incorporates the AD depicted in figure 2. The process produces nine scenarios from the activity depicted in figure 2 without iterating through the edge between the *Check PIN* control node and the *Get USER PIN* action node, more than the number of final nodes that exist downstream of the converging *Get USER PIN* action.

This ensures that the process only enters such an edge a limited number of times rather than risk being allowed to progress by a guard expression that may result in infinite recursion and hence, scenario explosion by our definition. Figure 5 represents the output from the process applied to the AD in figure 2.

For each of the ADs in the model being processed, an XML element tag is produced. The first tag identifies this as an AD and in this case the diagram name is "Create Session Activity" and the diagram's identifier is "di$6e85c53c:10566e36a51:-7388" are included as values in their appropriate attributes. The process then adds a scenario tag with an identifier inside the AD tags for each scenario that is developed. Within this opening and closing set of scenario tags the process then adds the sequence of elements that belong to each scenario. Each element in the scenario can be identified by type and contains other useful information belonging to that element.

This output is structured in such a way that it may be used for several purposes and in many different ways. For instance, it may be processed using XSLT (Extensible Style-sheet Language Transformations) to output the scenarios in HTML format for reporting purposes. Alternatively, it may be used to verify that usage-scenarios have been identified and documented during the requirements analysis phase and during regression design and development phases. In addition, it may be used to develop test-cases using the edge conditions and activity or local constraints that are included in a model. Therefore, the output of our process may become the input for various other useful purposes.

AD2US has processed several UML models and activity diagrams of various levels. Another example of the process is shown using the diagram depicted in figure 6, which is a hypothetical UC-based AD used for its complexity level rather than its implied functionality.

The UC depicted by the AD in figure 6 produces only two scenarios; even though it appears that there should be more. The fact is that control is passed from the Branch_1 control node to either Object_Flow_1 OR to Action_State_2.

Therefore, regardless of the concurrency portrayed by the fork and join nodes, processing can traverse only one of the two possible paths. The control node named "Branch_2" is actually a merge node which, unlike a join node, does not have to wait for a condition to be met before proceeding to its outgoing edge.

```
Begin Scenario Processing...
<ActivityDiagram name="Create Session Activity" id="di$6e85c53c:10566e36a51:-7388"/>
   <scenario id=0>
        <Initial id="sm$6e85c53c:10566e36a51:-6f48" name="Ready"/>
        <Edge id="sm$6e85c53c:10566e36a51:-6f31" name="" type="Edge"/>
        <ObjectFlow id="sm$6e85c53c:10566e36a51:-6f49" name="anATMcard"/>
        <Edge id="sm$6e85c53c:10566e36a51:-6f30" name="" type="Edge"/>
        <Action id="sm$6e85c53c:10566e36a51:-6f4a" name="get_cardDetails"/>
        <Edge id="sm$6e85c53c:10566e36a51:-6f32" name="" type="Edge" guard="[CARD_DETAILS_OK = false]"/>
        <Final id="sm$6e85c53c:10566e36a51:-6f44" name="Final_State_1"/>
   </scenario>
   <scenario id=1>
        <Initial id="sm$6e85c53c:10566e36a51:-6f48" name="Ready"/>
        <Edge id="sm$6e85c53c:10566e36a51:-6f31" name="" type="Edge"/>
        <ObjectFlow id="sm$6e85c53c:10566e36a51:-6f49" name="anATMcard"/>
        <Edge id="sm$6e85c53c:10566e36a51:-6f30" name="" type="Edge"/>
        <Action id="sm$6e85c53c:10566e36a51:-6f4a" name="get_cardDetails"/>
        <Edge id="sm$6e85c53c:10566e36a51:-6f2f" name="" type="Edge"/>
        <Action id="sm$6e85c53c:10566e36a51:-6f45" name="Get USER PIN"/>
        <Edge id="sm$6e85c53c:10566e36a51:-6f25" name="" type="Edge" guard="[USER.cancelled = true]"/>
        <Final id="sm$6e85c53c:10566e36a51:-6f42" name="Final_State_3"/>
  </scenario>
  <scenario id=2>
       ...
</ActivityDiagram>
```

Figure 5. formatted scenario output sample

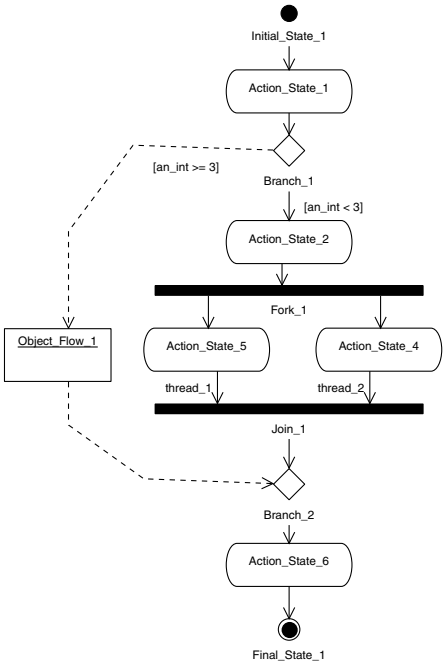

Figure 6. structured level activity diagram

However, the UML 2.0 superstructure rules that a *join* node must wait until all of its incoming edges have been finalised before its outgoing edge can be traversed. It does this using an **AND** condition in relation to the names associated with all of the join's incoming edges. Only when that condition is met, can the system enter, or put a token onto, the join's outgoing edge. Hence, we have a situation where no matter how many threads of operation are depicted within fork and join nodes, continuation is not allowed until each thread has completed.

When processing this situation, AD2US reaches the fork node and begins processing the threads one at a time until the join node is reached. Then, it processes the fork's next outgoing edge and so on until all threads are processed. When no more threads are left, AD2US then proceeds on the join node's outgoing edge.

AD2US first identifies the AD being dealt with, then it captures the initial node in the AD. The AD2US algorithm then locates the ID for the initial node's outgoing edge and then process this edge. The target of the first edge in the scenario directs AD2US to the Action_1 node. Following this procedure, AD2US eventually finds the Branch_1 control node, which has multiple outgoing edges. In the first scenario (id=0) AD2US traverses the outgoing edge from the branch node which directs processing toward the Object_Flow_1 node and onto the Branch_2 Merge node. The second scenario with an id=1 traverses the alternate edge toward the Action_State_2 node.

From this point AD2US continues to the join node where it processes this region in the manner described earlier in this section. It can be seen that as each thread in the concurrent region is finished, processing returns to the fork node and finds the next thread. When all the threads are completed, AD2US continues along the join node's outgoing edge, through to the Final_State_1 final node.

5 Conclusion and Future Work

The AD2US automated process of capturing the scenarios from ADs, offers a designer an efficient and effective method of producing usage based scenarios that can be used for validating whether all possible scenarios have been covered in the current design; and whether an initial test suite includes sufficient control and data flow coverage for all possible USs.

As a design evolves from UCs through other behavioural aspects of a proposed system to structural aspects, changes that are made to the behavioural design may effect activities or behaviour. The automatically produced set of scenarios, that result from our process, can be compared with the UC realizations which are created very early in the requirements gathering process, to determine whether latter design changes have created extra paths or possible scenarios that have not been included in data and control flow-based test cases and/or test suites.

In future work, we intend to use the output of this process to assist development of AD diagram completeness templates and in the production of test cases; this may be achieved using the guard, trigger and effect conditions that can be associated with the activity edges, as well as any activity or local action-based OCL constraints. In the future, as the output will be in XML-based structure, we may develop XSLT style-sheets to transformation the information into more designer friendly formats, for instance HTML.

References

[1] D. Amyot, X. He, Y. He, and D. Y. Cho. Generating scenarios from use case map specifications. In *3rd International Conference on Quality Software (QSIC'03)*, pages 108–115, Dallas, USA, 2003. IEEE Computer.

[2] X. Bai, C. P. Lam, and H. Li. An approach to generate the thin-threads from the UML diagrams. In *28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 546–552, Hong Kong, 2004.

[3] J. P. Barros and L. Gomes. Towards the support for crosscutting concerns in activity diagrams: a graphical approach. In *6th International Conference on the UML*, page 8, San Francisco, USA, 2003.

[4] B. Berenbach. The evaluation of large, complex uml analysis and design models. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 232–241. IEEE Computer Society, 2004.

[5] L. Briand and Y. Labiche. A UML-based approach to system testing. In *4th International Conference on the Unified Modelling Language (UML'2001)*, volume 2185, pages 194–208, Toronto, Canada, 2001. LNCS.

[6] A. Cockburn. Resources for writing use cases. WWW, 2005. Accessed Aug'05.

[7] A. Cockburn. Structuring use-cases with goals. WWW, ND. Accessed Aug'05.

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, USA, 1990.

[9] CutterIT. E-business test modeling with UML. WWW, 2001. Visited Aug 05.

[10] R. Eshuis and R. Wieringa. An execution algorithm for UML activity graphs. *Lecture Notes in Computer Science*, 2185:47–??, 2001.

[11] J. Heumann. Generating test cases from use cases. WWW, 2001. Accessed: August 05.

[12] M. Jarke. Scenarios for modelling. *Communications of the ACM*, 42(1):47–48, January 1999.

[13] P. Kamthan. Usage scenarios for uml diagram types, 2005. Accessed 19th August 2005.

[14] G. Kosters, H.-W. Six, and M. Winter. Coupling use cases and class models as a means for validation and verification of requirements specification. *Requirements Engineering*, 6(1):14, 2001.

[15] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 284–291, Busan, Korea, Nov.30 - Dec.3 2004. IEEE Computer Society.

[16] M. Liu, M. Jin, and C. Liu. Design of testing scenario generation based on UML activity diagram. *Computer Engineering and Application*, 2002(12):pp 122–124, 2002.

[17] OMG. OCL 2.0 - OMG final adopted specification. Standard - Technical Report ptc/03-10-14, Object Management Group, October 2003 2003.

[18] OMG. Unified Modelling Language, v2.0 superstructure. Standard - Technical Report ptc/03-08-02, Object Management Group, April 2004.

[19] R. Paul. End-to-end integration testing 2. In *25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, pages 286–290, Chicago, Illinois, 2001. IEEE.

[20] B. Regnell and P. Runeson. Combining scenario-based requirements with static verification and dynamic testing. In E. Dubois, A. L. Opdahl, and K. Pohl, editors, *Proceedings of the Fourth International Workshop on Requirements Engineering - Foundations for Software Quality (REFSQ'98)*, Pisa, Italy, 1998.

[21] SoftwareEngineer.org. A community for software engineers: Methods and techniques, 2005. Visited: 19th August 05.

[22] W. T. Tsai, X. Bai, R. Paul, and L. Yu. Scenario-based functional regression testing. In *25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, pages 496–501, Chicago, Illinois, Oct 2001. IEEE Computer Society.

[23] M. Zhang, C. Liu, and C. Sun. Automated test case generation based on UML activity diagram model. *Journal of Beijing Universit y of Aeronautics and Astronautics*, 27(4):pp 433–437, 2001.