Co-operative Education
Work Term Report

Path Traversal and Scenario Generation from Use Case Map

Name: XiangYang He
Student Number: 2417281
Employer: University of Ottawa
Supervisor: Dr. Daniel Amyot
Date Due: September 13, 2002

Table of Contents

Abstract

Use Case Map (UCM) is used in software engineering for describing functional requirements and high-level designs. The current software tool used in UCM is called UCM Navigator, it can be used to create and edit use case maps. My major work in this work term was to improve UCMNav to implement a new path traversal algorithm and to generate scenarios in XML format.

First, the background information about use case map is presented. Then, I analyze the problems existing in the original programs, propose a new algorithm, and discuss how the algorithm is implemented in scenario highlighting. Secondly, the neutral scenario definition is briefly discussed. Based on the new path traversal mechanism and scenario definition, I begin to work on scenario generation. The scenario generation algorithm is studied in depth, from algorithm analysis, class design to implementation details. Finally, some thought on future improvement regarding traversal on plug-in map and synchronization is presented.

Introduction

During this work term, I worked as a junior software developer for Professor Daniel Amyot at SITE, University of Ottawa. Professor Daniel Amyot is currently in charge of the standardization of User Requirement Notation (URN) at the International Telecommunication Union (ITU). The Use Case Maps (UCM) notation, as part of the URN is also being standardized by ITU.

UCM is used in software engineering for describing functional requirements and high-level designs, especially in the areas of telecommunications systems. UCM employs scenario paths to illustrate causal relationships among responsibilities. It provides an integrated view of behavior and structure by allowing the superimposition of scenario paths on a structure of abstract components. The combination of behaviour and structure enables architectural reasoning after which UCM specifications may be refined into more detailed models such as MSCs and UML interaction diagrams. [1]

The current software tool used in UCM is UCM Navigator, which was mainly developed by Andrew Miga at Carleton University. UCM Navigator is a fairly powerful tool, it can create and edit use case maps that are syntactically correct, perform path transformation and path connections based on the internal hyperedge representation, and provide support for scenario definitions.

However there are some problems with the tool: it can not traverse the maps completely, and sometime it may even crushes the program. Besides the algorithm it uses to output individual scenario definition is not the one suggested for standardization. My major task in this work term is to modify the UCM Navigator. It consists of three parts:

1) fix the bugs in the source code of UCM Navigator so that it can traverse scenario correctly
2) define a neutral representation of individual scenario in XML
3) modify the UCM Navigator to implement the new traversal algorithm and generate scenarios in the format defined in 2)

Considering the fact that the original tool has more than 80 thousand lines of source code, and it has been weakly documented, modification of existing program is not an easy task. I had to carefully read the code to try to understand the structure and algorithm the author used. Only based on the understanding of the program, can I find the problem it has and try to find ways to solve the problem. Furthermore it also requires me to grasp some basic concepts of Use Case Maps.

UCM and UCM Navigator

UCM Notation Elements

Use Case Maps notation was developed by Dr. R.J.A. Buhr at Carlton University. The notation is intended to be useful for requirement specification, design, testing, maintenance, adaptation, and evolution. It aims to bridge a modeling gap between requirement and design [2]. UCM paths are first-class entities that describe casual relationships between responsibilities, which are bound to underlying organizational

structures of abstract components. These paths represent scenarios that intend to bridge the gap between requirements and detailed design. [3]

The UCM notation is mainly composed of path elements, and components. The basic path notation addresses simple operators for casually linking responsibilities in sequences, as alternatives, or in parallel. Components can be of different nature, allowing for a better and more appropriate description of some entities in a system [3]. The basic path elements are:

- Start points: a scenario should have at least one starting points
- Responsibility: represents actions, functions to be performed
- End points: represents post-conditions and resulting effects.

Besides there are elements representing the relationship between paths, such as or-fork, and-fork, or-join, and-join, etc.

Figure 2 – 1 shows an example use case map where a user (Alice) attempts to call another user (Bob) through some network of agents [3]. In this map, req is the start point, ring and msg are the end points, vrfy, upd, and mb are responsibilities. Alice, Bob are components.

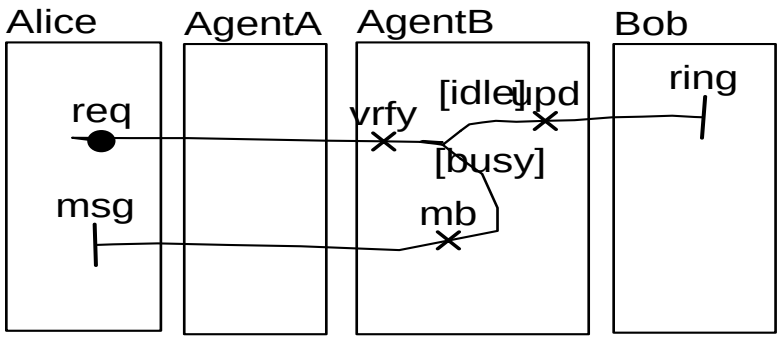

Figure 2 - 1

UCM Navigator

UCM Navigator (UCMNav) can support both UCM notation and XML format. Although it is a still a prototype tool, it is already robust enough for the creation and maintenance of UCMs. The path and component notation are fully supported. UCMNav ensures the syntactical correctness of UCMs manipulated, generate XML descriptions, export UCMs in Encapsulated or MIF format. Figure 2 – 1 displays a typical UCMNav interface.

The UCMNav has two main functions: managing all the logical objects that make up a UCM model and providing a visual interface that display the model and makes it possible to edit it. As such, the UCMNav classes can be divided into two major categories: logical classes and display classes. The logical classes store all the data associated with the model, while the display classes provide the user interface to access this data. [4]

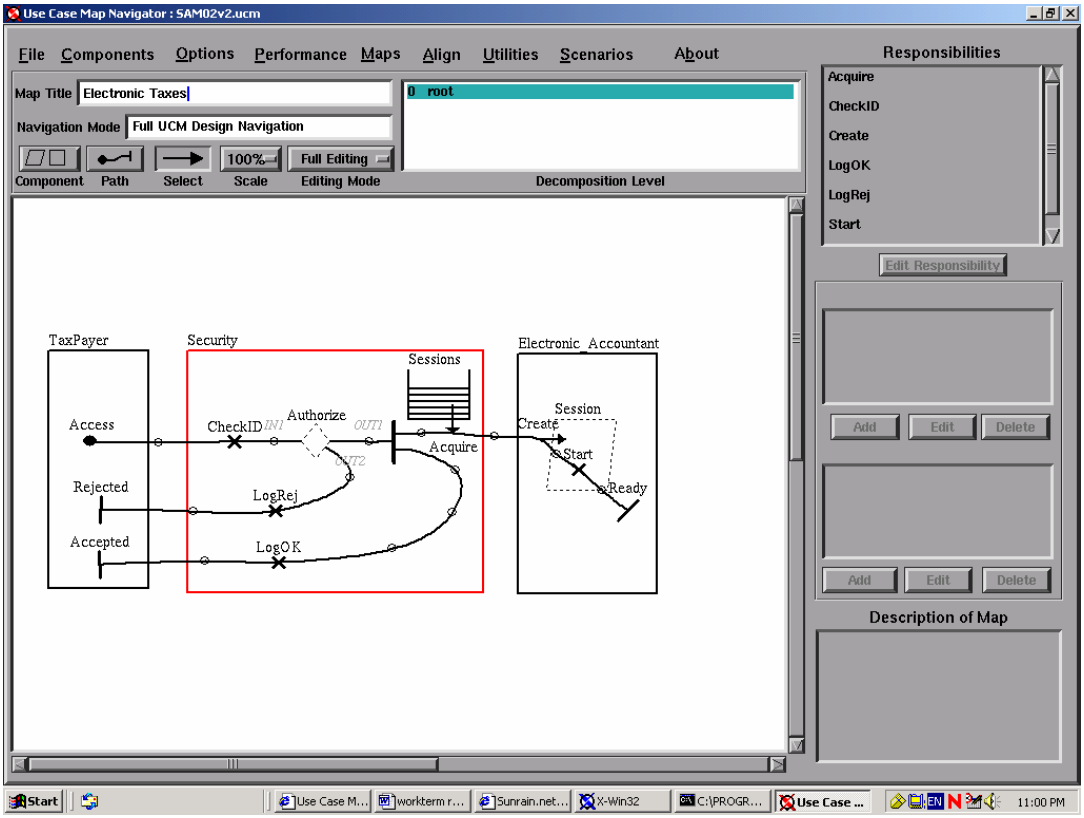

Figure 2 – 1

There are two main kinds of logical entities that we are concerned with in order to generate scenarios : path elements and components. UCMNav uses a hypergraph to represent the connections between UCM path elements in a path. A hypergraph is a graph whose hyperedges connect two or more vertices. It is composed of edges, called hyperedges, and vertices, or nodes. A hyperedge connects a set of multiple source nodes with a set of multiple target nodes. A node has a single hyperedge leading into it and a single target hyperedge leading from it. [4]

In the tool, the base hypergraph class is Hyperedge. It is a virtual class that defines all the methods and data that are common to every hyperedge. Every class with a single input and output is a direct child of Hyperedge. The MultipathEdge virtual class refines Hyperedge with methods to manage a variable number of multiple input or output paths. The OrFork, OrJoin, and synchronization classes are derived from Multipathedge.

Body

Software Tools

Through this work term I got acquainted with the some software tools. They greatly facilitated my job progress.

1. Source-Navigator

It is a source code analyzing tool. With it, programmer can edit source code, display relationships between class, functions, and members, and display call trees. It can also build projects with the authors own makefile, or by using Source-Navigator's build system to automatically generate a makefile [5]. It is especially useful for reverse engineering, since a lot of information, such as class hierarchy, function call relationship, can be extracted from it.

2. GNU debugger

The development platform of UCM Navigator is mainly in Linux. It provides the xxgdb symbolic debugger to enable programmers to analyze the execution of a program in terms of C++ language statements. gdb allows programmer to step through a program on a line-by-line basis while he/she examine the state of the execution environment. It also allows you to examine core files when a serious problem occurs. From the core file you can identify the line in the program while the failure occurred.

3. LiveDTD

LiveDTD is a program which scans through an XML Document Type Definition (DTD) to locate element and parameter entity definitions. Then it constructs an HTML version of the DTD with hot links from element references to element declarations, and from entity references to entity declarations. These links let users navigate through the DTD with ease. [6]

Task 1: Modify Path Traversal Algorithm

1. Application of Path Traversal Mechanism

A typical use case map may contain multiple scenario groups; each scenario group can also contain multiple scenarios. Thus it would be difficult to analyze one particular scenario in a complex map. UCMNav has a functionality called *scenario highlighting*. If user choose one particular scenario and press the "highlighting" button, the actual path traversal of this scenario will be shown in red or orange colors. The red color means that the path elements are traversed only once, while the orange represents that they are traversed multiple times.

If the scenario has no start points, or the variable for some branch condition is not initialized, or one of the Synchronization points, Timer, or Wait is not synchronized, the path traversal will fail and UCMNav will issue a warning message. So scenario highlighting can be used to analyze the scenario definition, check its syntactical correctness and completeness.

However the current UCMNav has some serious problems with regard to scenario highlight. It cannot produce the correct highlighting result when traversing some complex or unwell-nested maps, sometimes it highlights only one branch of the a and-Fork, and sometimes it may even crash the program.

Path traversal is the foundation for more advanced functionality of UCMNav, it has the application in the following areas: scenario highlighting, animation, MSC generation,

LQN generation, and test case generation [7]. Thus my first task will be to design an implement a new path traversal algorithm to highlight scenario.

2. Analysis of Existing Algorithm

The first step is to carefully examine the original source code, try to find the problem of the original design. Then I can design a new algorithm. The original algorithm is as follows:

1. Get the start point of the scenario
2. Process this hyperedge
3. Get the next hyperedge following it
4. check if the hyperedge is NULL, if Yes go to step (1); if No go to step (2) until the last of the scenario start point is reached

At a first glance, this algorithm seems have no problem. It just iteratively get the next available hyperedge, process it, and then look for the next following it.

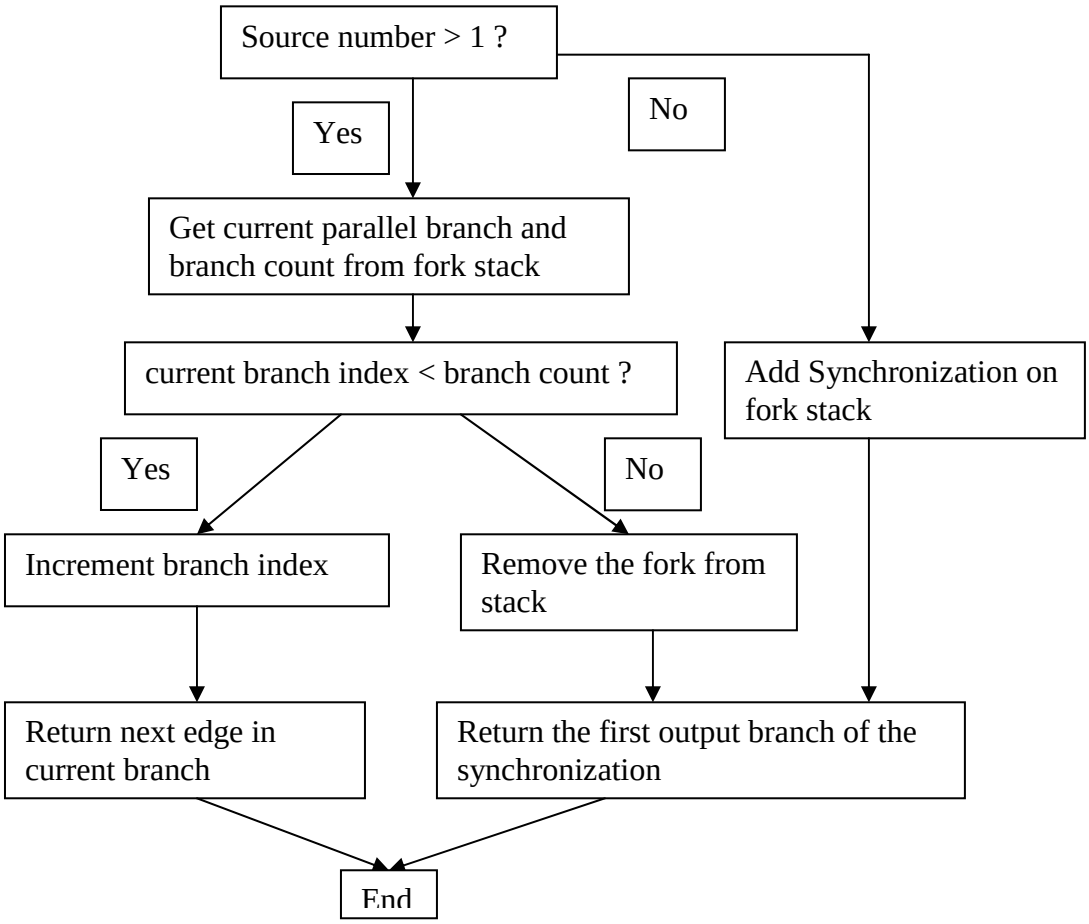

Figure 3 – 1

The method to process the hyperedge is called ScanScenarios, which is a virtual method, each subclass of hyperedge has its own implementation of the method. Through carefully trace some example UCM maps, I found that the problem resulted from the implementation of method ScanScenarios in class Synchronization, Wait, Timer, and Stub. Take the synchronization's ScanScenarios method as an example, the algorithm is shown in figure 3 -1.

When current branch index equals to branch count, the algorithm would assume that the all branches of the synchronization point have been visited, and quietly return the next hyperedge following the synchronization. For well-nested map , this is the case, but when map is unwell-nested, then there is a problem. Let us exam the map below and take a closer look at how the algorithm works.

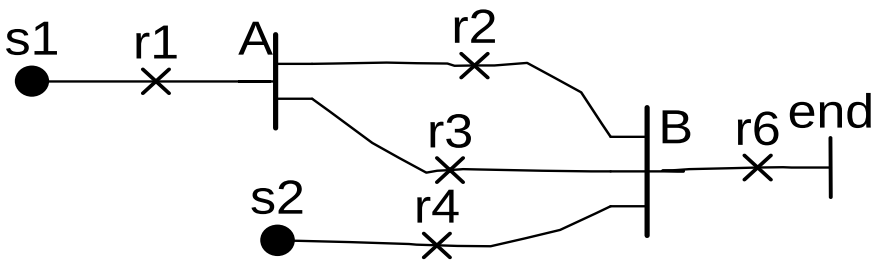

Figure 3 - 2

The above map have 2 start points S1, S2, 1 end point, and 2 synchronization points, synchronization A has 2 output branches, synchronization B has 3 input branches. If the scenario starts from S1, it will traverse in the procedures below:

1. process s1, go to next hyperedge r1
2. process r1, go to next hyperedge A
3. put A on fork stack, process the first output branch r2, go to next hyperedge B
4. peek the fork stack, get the next branch r3
5. process r3, go to next hyperedge B
6. peek the fork stack, since every output branch of A has been traversed, pop up A from stack, and get next output branch r6 of synchronization B
7. process r6, get next hyperedge end point end
8. Since there is still scenario start points left for visit, get the next start point s2
9. process s2, get next hyperedge r4
10. process r4, reach next hyperedge B
11. peek the fork stack, it is now empty, and return a NULL pointer, the program tries to manipulate on the NULL pointer, and thus crashes at this point.

Obviously, although 2 branches of A have been visited, we can not judge from it that B has been synchronized. Use Case Maps can be in any complex combinations. That is where the problem resides.

Similar problems exist for class Timer, Wait, both are subclasses of Wait_synch. When a Wait or Timer is encountered, it will call a method NextScanningPath in class MSC_generator, which tries to find the hyperedge that triggers the Wait_synch. But the method NextScanningPath only restrict itself to the top element of the fork stack while looking for the trigger element. If one of the branches of the top fork triggers the Wait_synch, that is fine. Otherwise it returns the next starting points, as though it will trigger the Wait_synch. This is not true in most cases.

3. Design of New Traversal Algorithm

From the above analysis, we conclude that the existing traversal algorithm has the following problems:

1. It assumes that UCMs are in well-nested form.
2. It treats Synchronization, Timer, Wait in different ways during traversal, thus makes the program very complex.

In order to solve the problems, I propose a new depth-first traversal mechanism. The detailed algorithm is as follows:

Variables:

Fork_Stack : holding the and-fork elements, if all branches of the fork have been visited, the fork will be popped up from the stack

Path_Elements: stores all Synchronization, Wait, Timer elements when they are encountered for the first time during traversal

Algorithm:

1. Start from one of the start points defined by the user.
2. Moves from path element A to path element B if continuation condition for element A is met.
3. If a Synchronization (in particular and-join), Timer, or Wait is encountered, register this element with Path_Elements, then get the next parallel branch from the top of the Fork_Stack. If all branches of the top element have been visited, remove it from Fork_Stack.
4. If the Fork_Stack is empty, get the next start points, and continue.
5. If one of the Synchronization, Timer, or Wait has been synchronized (that means that all input branches of it have been visited), mark the corresponding element in Path_Elements as all visited, and get the first output branch of this hyperedge.
6. If an end point is reached, first check the Fork_Stack, if it is not empty, then get the next parallel branch, otherwise get the next start points, and continue.
7. After no more start points of the scenario are left for traversal, check each element of Path_Elements, if any one of them is not marked as all visited, that means there is an error with the scenario definition, generate an error message and abort traversal. Otherwise traversal succeeds.
8. Clean up the Stack and reset the generation state of all elements in Path_Elements.

Since the UCMs can be in well-nested and unwell-nested form, and Synchronization and Wait_synch share some similar characteristics, the algorithm adopts a depth-first approach, it keeps traverse the path elements until a stop point (means and-join, Wait, Timer) is reached, then it backtracks to get next available hyperedge. The hyperedge can

be either one branch of an and-fork or one of the scenario start points. So this approach treats Synchronization, Timer, and Wait in similar way while traversing, and thus is much simpler and robust than the original one.

4. Implementation

The implementation of this algorithm is mainly accomplished through modifying existing source code in class msc_generator.cc, msc_generator.h, synchronization.cc, synchronization.h, timer.cc, timer.h, wait.cc, wait.h, and class node.cc, node.h, etc.

First I removed the functionality of scenario highlighting from method ScanScenarios in class Hyperedge and its subclasses. The original method tries to accomplish scenario highlight and MSC generation in one method, thus make its structure too complex and error phony. Then I added a method HighlightScenario to Hyperedge and its subclasses to accomplish scenario highlighting through implementing the new algorithm. In addition, I also made the following modifications:

1. In class node.h, add an instance variable bool visited to keep track of the visiting state of the node, in class node.cc add corresponding set and get method, and in the constructor set the variable's value to false.
2. In class empty.cc, in the method HighlightScenario add a function call to set the target node's variable visited to true.
3. In class map.h, add instance variable scanning_parent_stub with the type Stub * to keep track of the parent stub of current map, and initialize its value to NULL in the constructor, and add corresponding set and get method.
4. In class stub.cc, in method ScanPlugInMap add a function call to method SetScanningParentStub to variable submap.
5. modify the virtual method HighlightScenario for each subclass of class hyperedge.

After the coding phase, I thoroughly tested the program on many UCM examples. It works well.

Task 2: Define Neutral Scenario Definition

The existing UCMNav is a fairly complex tool, it has many functionalities, such as scenario highlighting, generating MSC, generating LQN, export maps in SVG, CGM, EPS, and MIF format, as well as the basic file operations. So the source code is very complex, it is difficult for maintenance. Especially, considering the fact that the UCMs may late be transformed to TTCN, LOTOS, and even UML, should we add all these functionalities to UCMNav? The answer is No. It would be much easier to generate a neutral representation of UCM scenarios, and have other tools to process the scenario to output TTCN, LOTOS, or UML, etc.

My second task is to define a new scenario definition in XML. It should provide adequate information for future processing of scenarios. This task was completed with the help of Dr. Daniel Amyot.

The top element of the scenario is scenarios, it may contain multiple scenario groups, and each scenario group may also contain multiple scenarios. The path progression of a scenario is the most important part of scenario definition. There are 2 types of path progression, sequence (seq) and parallel (par). The most basic elements of scenario are do and condition. do represents responsibility, start point, end point, wait, and timer. condition represents path selection condition in or-fork. For detailed definition of scenario, please refer to Appendix 1: Scenario Documentation Type Definition.

Task 3: Generate Scenario from UCM

1. Scenario Generation Algorithm

Scenario generation consists of two major aspects, one of them is scenario traversal, which is basically the same as path traversal algorithm in Task 1; the other one is scenario restructuring. In this section, we will focus on the algorithm used in restructuring scenario.

The scenario is defined in XML, which is a well-nested format, however the actual UCM maps can have any complex format, and they may be unwell-nested. Therefore during the transformation of UCM maps to new scenario we need to make some modifications to the representation of UCM. The transformation process is called collapsing, it takes place when the waiting place (such as Synchronization, Wait, Timer) is synchronized. We will use the example in Figure 3 – 3 to illustrate the collapsing algorithm.

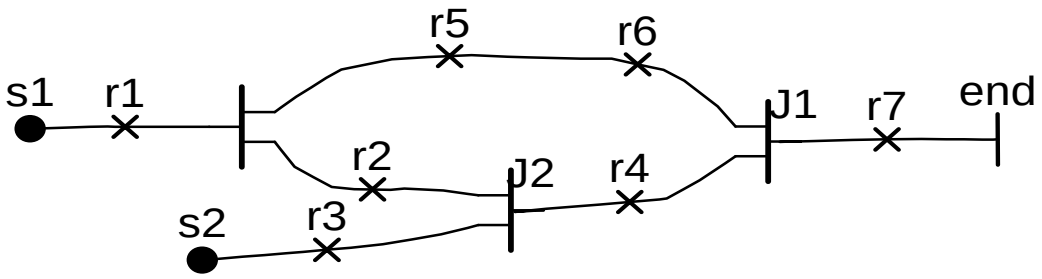

Figure 3 - 3

The scenario traversal starts at start point s1, and continue to r1, then it reaches the first and-fork. At the pint, one parallel path r5 is chosen, and the fork is put on a stack. The traversal continues until it reaches a synchronization point where the traversal gets stuck. Then it pops up the stack and get the next parallel branch r2, the traversal gets stuck again. Since the stack is empty, the path traversal mechanism will try to get the next start point, in this case s2, and start traverse from s2. The traversal continues to r3, after that it reaches synchronization point J2. At this point, all input branches of J2 have been visited, so J2 is synchronized. The parallel paths need to collapse now. Before collapse, the paths are shown in Figure 3 – 4.

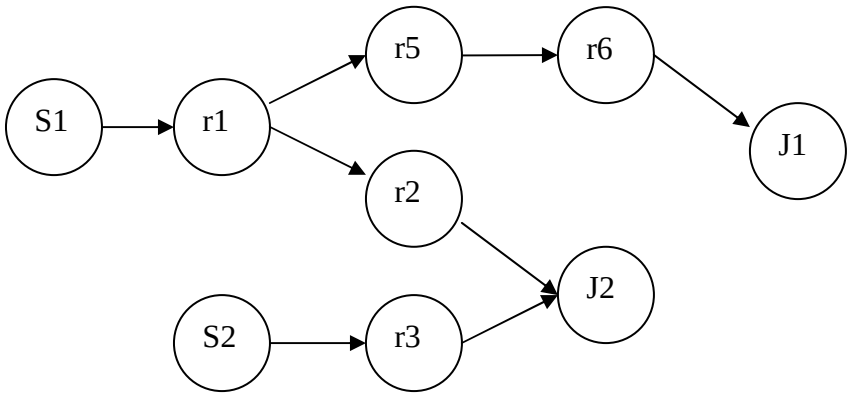

Figure 3 - 4

On this graph, each node represents a start point, responsibility, or an and-join. To collapse the paths, we should search through the 2 branches starting from synchronization point J2, for each node, if it has next node that do not leads to J2, we will move the next node to J2, until the start point is encountered, or the common ancestor of the 2 branches is reached. Since the 2 branches all start from start points, they have no common ancestor. In order to make the parallel block easy to process, we add an additional node S before start points s1 and s2. So this additional node becomes the start point of the parallel. After collapsing, a parallel block starting from S and ending at J2 is constructed; the paths are shown Figure 3 – 5.

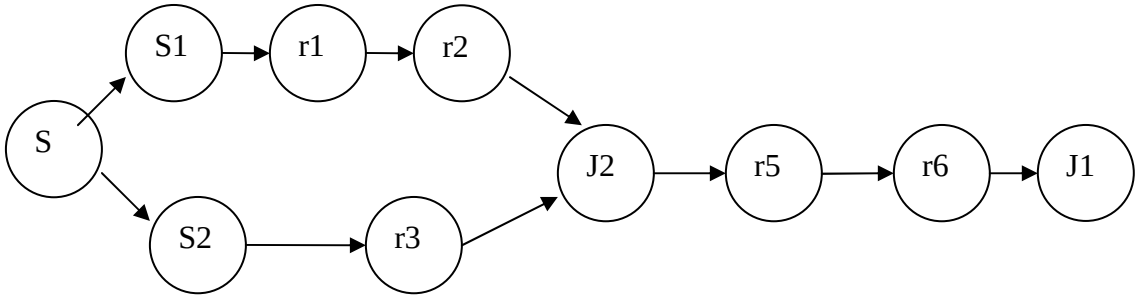

Figure 3 - 5

Since the transformation process is a partial ordering, so after collapsing, some of the scenario information is lost. For example, in the original UCM map, r5 should start after r1 and before r6, but in the collapsed graph, r5 is deemed to start after J2.

The path traversal resumes from J2, continues to r4, until it reaches J1. At this point J1 is synchronized, so it needs to be collapsed again. Before collapsing, the graph is shown in Figure 3 – 6.

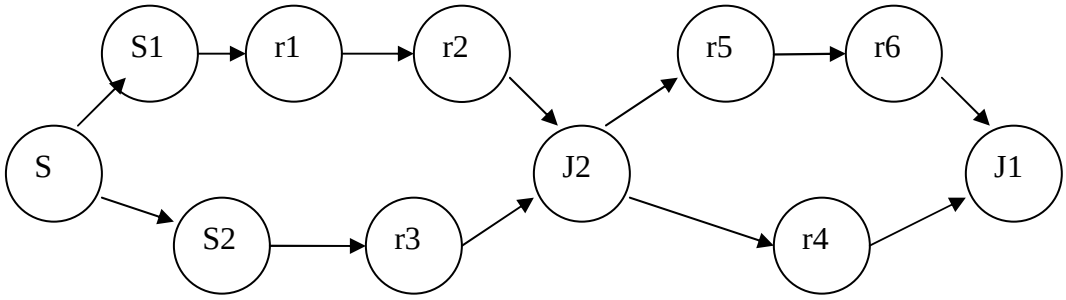

Figure 3 - 6

Since the 2 branches of J1 are both starting from J2, J2 is the common ancestor of the 2 branches. For each node of the 2 branches, we should check if any one of its next nodes does not lead to J1, if not, we should shift it to J1, this process continues until it reaches J2. In this particular case, no node needs to be shifted. So the graph after collapse is essentially the same as the one before collapse.

Path traversal continues from J1, through r7, to the end point end, since the stack is empty, and there is no more start point left to be traversed. The path traversal ends at this point. The final scenario is shown Figure 3 – 7. For the output of the scenario in XML format, please refer to Appendix 2: Example Scenario Output 1

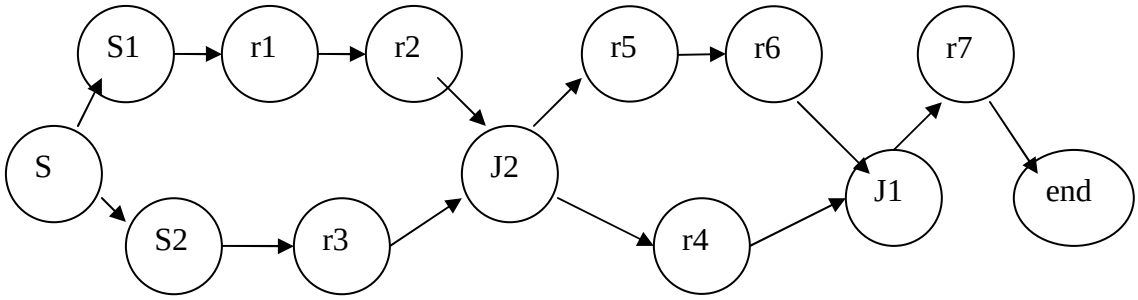

Figure 3 - 7

2. Design of Classes

The new scenario definition requires us to make modifications, such as collapsing to the maps while traversing the maps. However we should not change the original map, so we need to design some additional classes to store the path information, to make necessary processing, and leave the original map intact.

The scenario is actually a nested graph, it consists of nodes, and each node represents a responsibility, start point, end point, and-join, and condition. The node should also have pointers pointing to its next nodes and previous nodes. Each node can point to multiple next nodes and previous nodes. Since do and condition are two different identities in the scenario definition, they have different types of attributes, it is reasonable to have 2 classes to represent the two types of objects.

Besides, the two classes also share some common features, such as they both have a field to represent the hyperedge-id of the hyperedge, more importantly, each of them should have pointers pointing to its previous and next nodes. It is appropriate to have a super class to stand for the common attributes of element do and condition.

Based on the above analysis, I designed three classes to represent the elements of the scenario definition. The class hierarchy and major data field are listed in Figure 3 - 8.

Class Path_data stands for the common characteristics of path element, it has the methods necessary for list operations, such as add an element after the element, add an element before the element, remove an element from previous or next list. The instance variable scanCount is not an attribute of the class Path_data, but it is used during the process of collapsing, which will be illustrated in next section.

Class Action is a subclass of class Path_data. It corresponds to element do in the scenario definition. Class action contains methods necessary to initialize, destruct the Action object, and the methods to print the scenario element in XML format. Similarly, class Conditions is also a subclass of class Path_data, which corresponding to element condition in the scenario definition.

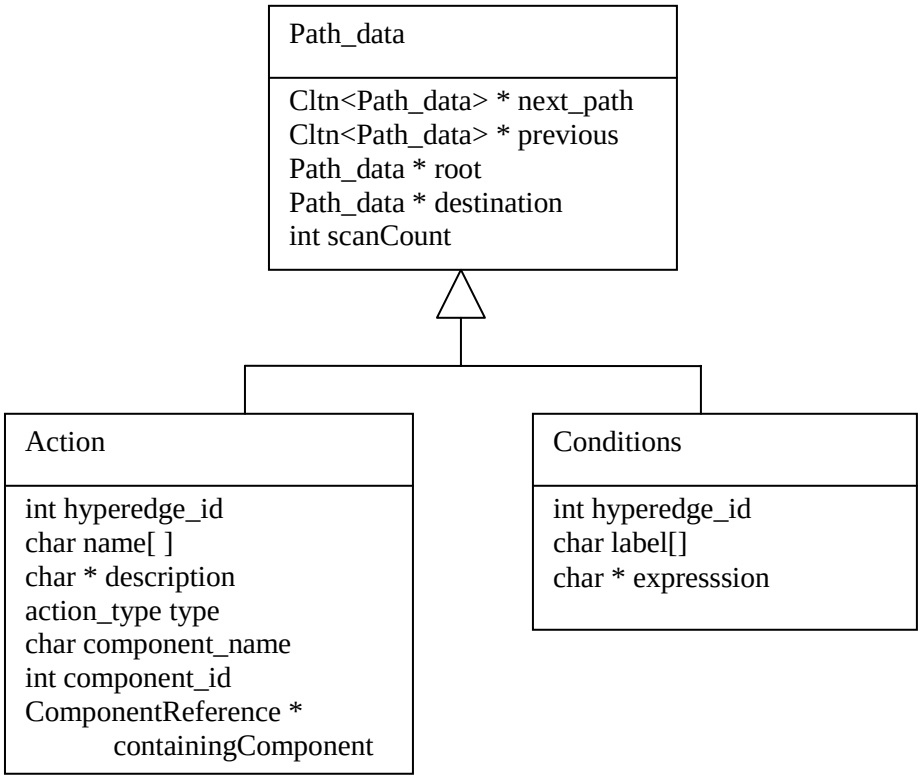

Figure 3 – 8

There is also a class SCENARIO_Generator, which is used to initializing, managing the scenario generation process, as well as output the scenario in XML format. Class SCENARIO_Generator is very similar to class MSC_Generator. It has one additional variable Cltn<Path_data *> path_traversal to hold the Path_data elements it have created during the process of traversal. When it outputs the scenario information, it just prints the path_traversal one by one.

3. Implementation Details

From the scenario generation algorithm presented in section 1, we know that the actual collapsing takes place after the path traversal synchronizes an and-fork, a Timer, or a Wait. Since Timer and Wait have much characteristic in common, I decided to implement the collapsing process at class Synchronization, Wait_Synch respectively. Wait_Synch is a super class of Timer and Wait.

Collapsing is a complex process; it involves finding of the root element, moving the redundant element, etc. Especially for Synchronization, it can have multiple input branches, which makes the structure of maps very complex. We will take a close look at the methods in Synchronization.

Every time when a synchronization point is synchronized, it means that a parallel block can be formed. Since there is no parallel object in our class definition, we have to define the scope of the parallel through its root and destination. Root is the parallel's start point; destination is the parallel's end point. For the second parallel block in the example of Figure 3 -3, the root is J2, and destination is J1. Path_data J1's instance variable root should be set to J2.

The method SearchRoot in class Synchronization is intended to find the root of the parallel block. In order to find the root, we should backtrack along each input branch of the synchronization, and increments the instance variable scanCount of each Path_data element along the path until the end of the path. At the end, the first element with the scanCount greater than 1 is the root of the parallel block. If there is no element with scanCount greater than 1, then it means that all input branches starts with different start points. For example, in figure 3-3, the first parallel block with destination J2, the 2 branches are S1 - r1 – r2 and S2 – r3, each element's scanCount equals to 1. So we should set the J2's destination equal to NULL.

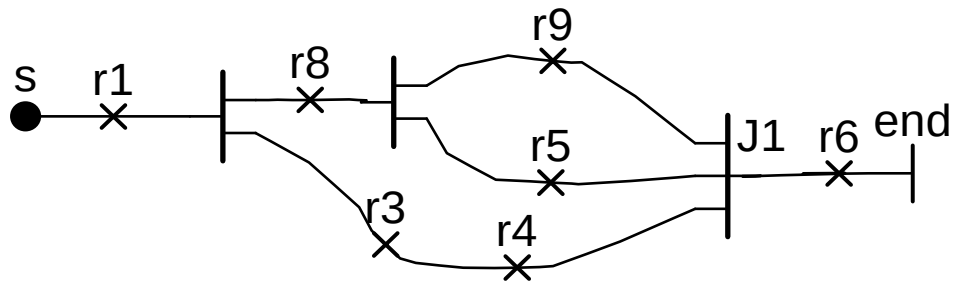

Figure 3 – 9

The reason we use scanCount to find root element and Boolean variable incompleteSearch in method TraverseScenario is mainly to deal with the situation depicted in Figure 3 – 9. When path traversal synchronized and-join J1, there are 3 input branches, obviously, r9 and r5 should form an inner parallel block (let's denote it P1), P1 and r8 form one branch of the outer parallel block, r3 and r4 form another branch.

After backtracking from J1 to increment scanCount, r8 has a scanCount value 2, S and r1 with value 3. Since r8's scanCount value is less than the number of input branches of J1, we can conclude that there exists an inner parallel, variable incompleteSearch is set to true, and r8 becomes the root of inner parallel. So we need to collapse the map for the inner parallel first, after that we will collapse the out parallel block.

When performing collapsing of inner parallel, we need to insert an additional and-join between the inner parallel and J1 in order to facilitate collapsing of outer parallel. Before collapsing of inner parallel block, the paths are shown in Figure 3 – 10.

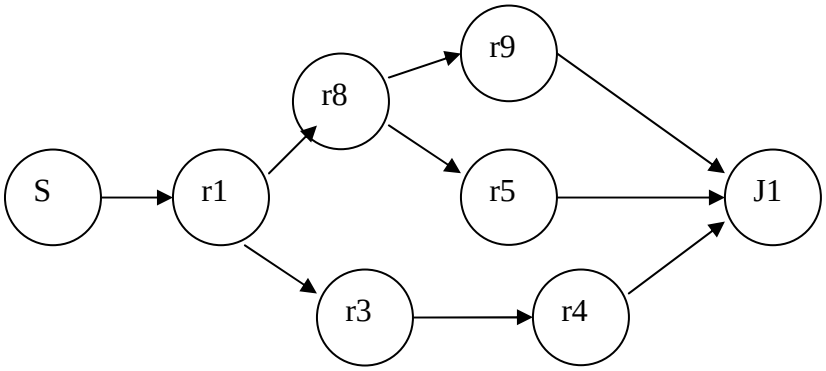

Figure 3 – 10

After collapsing the inner parallel, supposing that the additional and-join is called JJ, the new figure is shown in Figure 3 – 11. From the figure, we can see that r8 is the root of the inner parallel; JJ is the destination of the inner parallel. Now the total number of input branches of J1 becomes 2.

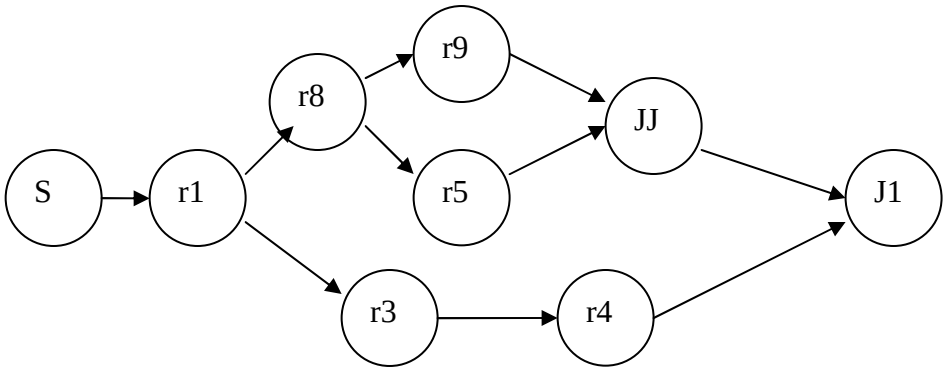

Figure 3 – 11

Since variable incompleteSearch is true, it means there exists outer parallel remain to be collapsed. So we should backtrack from J1 to increment scanCount, and search the new root again. This time r1's scanCount is 2, which is equal to the number of input branches of J1, so r1 becomes the new root, J1 is the destination of the outer parallel, and variable incompleteSearch should be set to false. The map is already well-nested, so the second collapse does not modify the graph. Because incompleteSearch is false, the collapsing of J1 is complete, and we can continue to traverse the map. The final map is shown in Figure 3- 12. For the output of the scenario in XML format please refer to Appendix 3: Example Scenario Output 2

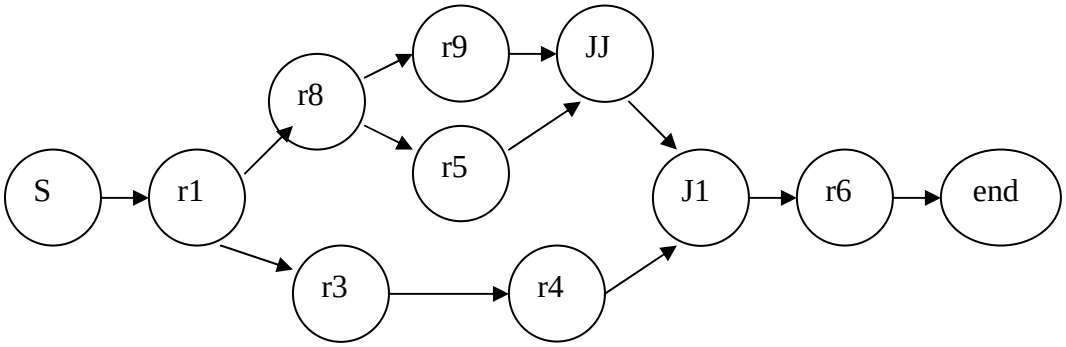

Figure 3 – 12

Discussion on Future Design

Although the functionality of path traversal and scenario generation can work properly most of the time, there are still some issues that need to be solved in order to improve the algorithm.

Reset of Synchronization Point

With current path traversal algorithm, every time when a synchronization point is synchronized, the algorithm will reset Boolean variable visited to be false for all input branches of the synchronization. This has two limitations, first, it cause the traversal result tightly coupled with the order of start points; second, there may be situations, that the synchronization point can be kept synchronized once it was synchronized. Let's first look at the example shown in Figure 4 – 1.

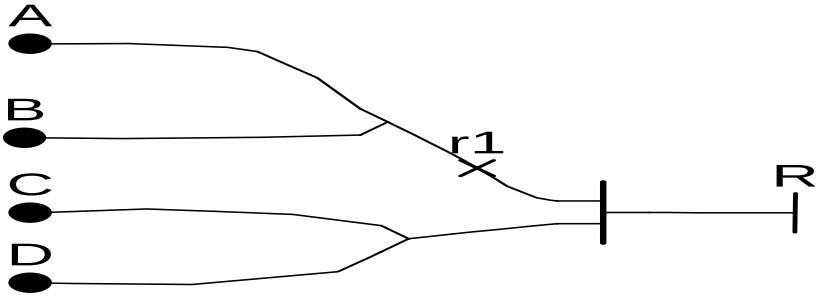

Figure 4 – 1

For this example, when the start points of one scenario begin in the following order: A-B-C-D, start points A, B, C first triggers the and-join, path traversal continues to R, then start point D begins to be processed . When it reaches and-join, the path traversal will generate an error message because only one branch is visited.

However when the start points begin in the order of A-C-B-D, A, C first triggers the and-join, path traversal continues to R, then B, D begin to traverse, and they will trigger the and-join again, so the synchronization is actually traversed two times.

Therefore, I think that we can add an attribute to Synchronization, so that the designer can specify if he/she wants to keep the synchronization open (synchronized) once the synchronization is triggered. For example a Boolean variable keep_open, if it is true, then after the synchronization was synchronized, we do not need to reset the input branches' visited value; if it is false, then we should reset them.

Traversal on Plug-in Map

If path traversal visits a plug-in map more than once, is it visiting the same plug-in or a new instance? [5] This is a question still not answered. Current traversal algorithm does not work when a plug-in map is bound with multiple stubs in one root map.

The reason I decided not to consider this situation, is that there are many plug-in maps in a complex UCM map, and each map may contain multiple start points. If the start points in plug-in map are bound with the stub entries, then it is not difficult to implement an algorithm to determine which plug-in map is being visited. However the current UCM syntax allows the maps at all levels to have start points and the start points in plug-in map do not have to be bound with the stub entries, it is virtually impossible for the current path traversal mechanism to determine which stub's plug-in map is being visited when a start point in a plug-in map is initiated. Again let's look at an example in Figure 4 – 2.

Here we are supposing that when traversing a plug-in map it is a new instance of the map being traversed. The map in Figure 4 – 2 (a) is a root map, it has two stubs, both of the stubs are bound to the plug-in map in Figure 4 – 2 (b). Besides, start point s1 in plug-in map is bound to the stub entry of both stubs; end point e1 is bound to the stub exit path of stubs. The traversal procedure is as follows:

Start point s, and responsibility r1 are visited. We reach the and-fork, the fork is put on the stack, and the first output branch s1 is chosen. s1 and r3 are visited. The traversal reaches the and-join of plug-in map, and gets stuck. So it pops up the stack and gets the next parallel branch – s1 of plug-in map bound with stub st2. Path traversal continues to r3, and then it gets stuck at the and-join. Since the stack is empty, the traversal mechanism tries to get the next start point of the scenario. So s2 is picked up, now we have the problem, is this s2 belonging to the plug-in map of stub st1 or st2?

Currently, the traversal algorithm uses a variable decomposition stack to determine the stub decomposition level. When an input branch reaches its synchronization point, wait, or timer, the decomposition levels of different branches are first compared to determine if

they belong to the same instance. It assumes that all branches of the same instance should have the same decomposition level and decomposition stack. However the start point in plug-in maps completely destroyed the mechanism, we have no way to figure out which decomposition level the start point is at.

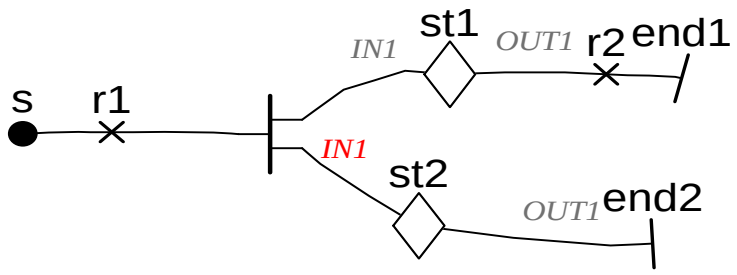

Figure 4 – 2 (a)

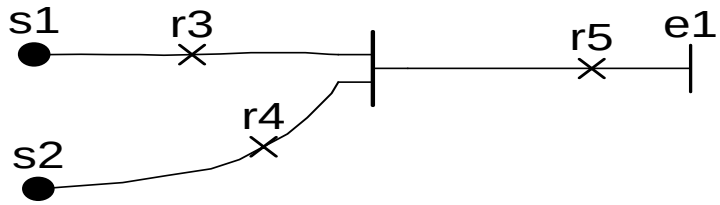

Figure 4 – 2 (b)

Thus the decomposition level and decomposition stack becomes redundant under current circumstance. I did not delete the two variables, because many issues are undetermined around plug-in map and stub, we may need these structures later.

If no plug-in map is allowed to be used more than once in the same root map, then we can completely remove the decomposition level and decomposition stack. If multiple instances of the same plug-in map are allowed, and all start points should begin from the root map, then we can slightly modify the algorithm to make it work well. Otherwise we would have to find a different approach to determine which instance is being traversed.

Conclusion

Use Case Map is mainly used in software engineering for describing functional requirements and high-level design. UCMNav is currently the only tool used to create and edit use case maps. The major objective of this project is to modify UCMNav to implement a new path traversal algorithm and generate scenarios in a neutral scenario definition.

After 3 months hard work, and with the help of Professor Daniel Amyot, I have successfully completed the project. This report presents the design and implementation of

the algorithms I have adopted. First, it analyzes the original path traversal algorithms and point out the problems with it. Then, it proposes a new traversal algorithm, and illustrates how the algorithm is implemented with examples.

Scenario generation is a new functionality added to UCMNav. It is used to generate scenarios under new scenario definition, so that the scenarios can be post-processed to generate MSC, TTCN, and LOTOS. Scenario generation algorithm is addressed in detail, especially the process of collapsing the maps.

Finally, it raises some questions in traversing synchronization and plug-in maps. It discusses some design options for each question and how they will be solved in future designs to improve the functionality of UCMNav.

References

1. Daniel Amyot, Draft Specification of the Use Case Map Notation (Z.152), [on-line] Available: http://www.usecasemaps.org/urn/urn-meetings.shtml#latest, 2002

2. R.J.A. Buhr, R.S. Casselman, Use Case Maps for Object-Oriented Systems, Prentice Hall, 1996

3. Daniel Amyot, Use Case Map quick tutorial version 1.0, [on-line], Available: http://www.usecasemaps.org/pub/UCMtutorial/UCMtutorial.pdf, 1999

4. Dorin B. Petriu, Layered Software Performance Models Constructed from Use Case Map Specifications, M. Eng thesis, Carleton Univ., [on-line] Available: http://www.usecasemaps.org/pub/dp_msc.pdf, 2001

5. Source Navigator documentation, [on-line] Available: http://sources.redhat.com/sourcenav/, 2001

6. LiveDTD documentation, [on-line], Available: http://www.sagehill.net/livedtd/manpage.html, 2000

7. Daniel Amyot, UCM Scenarios and Path Traversal, [on-line] Available: http://www.usecasemaps.org/urn/200203-geneva/UCM-Scenarios-Traversal.ppt,