Co-operative Education
Work Term Report

Improving The Generation Of Scenarios From Use Case Map Specifications.

Name: Shuhua Cui
Student Number: 2442048
Employer: University of Ottawa
Supervisor: Dr. Daniel Amyot
Date Due: May 16, 2002

Abstract

Use Case Map (UCM) describes functional requirements and high-level designs with causal paths superimposed on a structure of abstract components. The UCM supports to model both structural and behavior aspects of the proposed system. The generation of individual scenarios from UCM specifications promotes the validation of requirements and the transition from requirements to design. The current software tool used in UCM is the UCM Navigator that is capable of producing a scenario definition file (in XML format). Based on the XML file, transforming the XML scenarios into other scenario languages (i.e., Message Sequence Charts) has been achieved by a converter using XSL transformation. Initially, the main goal of my work term is to extend the work and generate UML (Unified Modeling language) Sequence Diagrams (represented in XMI) from the scenario definition files. However, through three weeks research, all known UML tool don't support XMI, so my major work in this work term is changed into improving the generation of Scenarios from Use Case Map Specifications. This includes improving the UCMNav to generate UCM scenario in correct XML format and modifying the converter to implement an extended transformation algorithm.

TABLE OF CONTENTS

CONTENTS

INTRODUCTION **4**

USE CASE MAP **5**

CONCEPTS AND UCM NOTATION ELEMENTS 5
UCM NAVIGATOR 6
SCENARIO DEFINITION 7
SCENARIO DEFINITION ELEMENTS 7

MESSAGE SEQUENCE CHART **8**

TRANSFORMATION **8**

BODY **8**

TASK 1: EVALUATE UML TOOLS FOR XMI SUPPORT 8
TASK 2: IMPROVE UCM NAVIGATOR 9
Software tool *9*
The problems with current UCMNav *9*
The solution for above problems *9*
Implementation *10*
TASK 3: MODIFY THE ORIGINAL XSL TRANSFORMATION ALGORITHM 11
Environment and tools *11*
Analysis of original XSL transformation algorithm *11*
Extension of transformation algorithm *13*
Implementation *14*

DISCUSS ON REMAINING ISSUES **14**

CONCLUSION **15**

REFERENCES **16**

APPENDIX **17**

APPENDIX 1: MODIFIED SCENARIO DOCUMENT TYPE DEFINITION **17**
APPENDIX 2: AN SCENARIO OUTPUT AFTER THE UCMNAV IS MODIFIED 18

Introduction

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

UCM graphical models describe functional requirements and high-level designs with causally linked responsibilities, superimposed on structures of components. The UCM is used to model behavior and structure of a proposed system .The combination of behavior 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 (a multi-platform tool written in C++). It is mainly work of Andrew Miga at Carleton University. This tool can highlight the UCM path traversal according to scenario definitions and generate other representation of individual scenarios like Message Sequence Charts in textual Z.120 format. Initially, path traversal and generation of MSCs are combined into one algorithm. This makes the programming difficult to maintain due to complexity. To remedy the situation, a novel two-step approach to the generation of scenarios from UCMs is presented [4]. This approach is very robust, flexible, and extensible because it decouples the UCM path traversal algorithm from the generation of the target scenarios and hence avoids the complexity of the algorithms. In the first step, a newly developed traversal algorithm in UCMNav 2.1 can generates XML scenario files Based on the XML file [10]. In the step 2, a converter is used to transform the XML scenarios into other target languages (i.e. MSCs) using XSL Transformation (XSLT, [11]). “XSLT provides the best solution for transforming XML files to various textual representations because it supports overriding mechanisms which allow one to substitute default transformations with customized ones for particular contexts. Such refinement is usually needed to express behavior details that are not found in UCMs (i.e. specific messages). Many XSL specifications can be used on the same XML scenario file to provide transformations to various target languages” [4]. Figure 1 gives an overview of this scenario generation process.

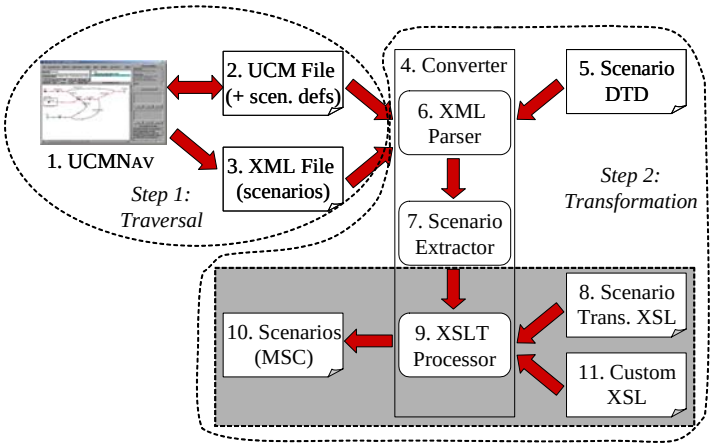


Figure1: Overview of the Scenario Generation Process [4]

Initially, The main goal of my work term is to extend the work and generate another target scenario representation: UML (Unified Modeling language) Sequence Diagrams in XML Metadata Interchange (XMI) from the scenario definition files. However, through three weeks research, all known UML tool don't support XMI, So my major work in this work term is changed into improving the generation of Scenarios from Use Case Map Specifications. This includes improving the UCMNav to generate UCM scenario in correct XML format and modifying the converter to implement an extended transformation algorithm.

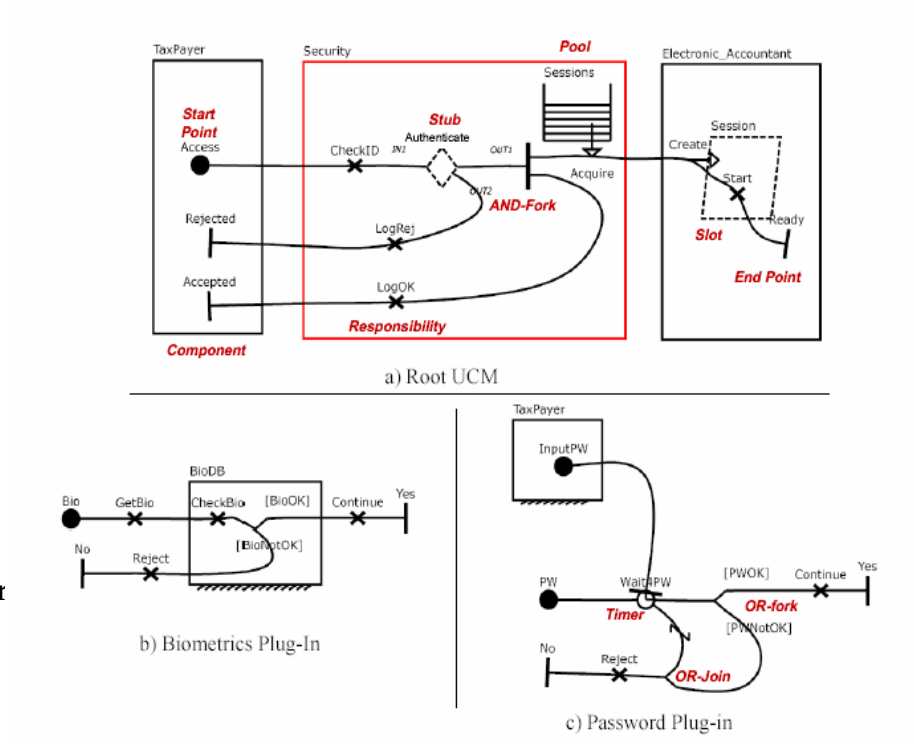
My first Task in this term is to evaluate the UML tools for XMI.

My second task in this work term is to improve the UCM Navigator to generate UCM scenarios in correct XML format. It consists of four parts:
1. Generate component role attribute for do element
2. Generate component name correctly for responsibility in plug-in
3. Generate condition element for plug-in selected in a dynamic stub.
4. Generate component attribute for par element

My third task in this work term is to modify the converter to implement a extended transformation algorithm. It consists of four parts:
1. Environment instances (or endinstances) need to be generated if the start point or the end point are not bound to a component
2. The messages coming from or going to the environment need to be generated
3. In parallel, inner-instance messages between component need to be generated correctly
4. Messages generation after an AND-Join / MSC par inline statement

Use Case Map
Concepts and UCM Notation Elements

Figur



Use Case Map (UCM) is a scenario-oriented notation for modelling functional requirements. The UCM notation provides a way to express and validate high-level architectural designs of concurrent real-time systems [2].

The scenarios are (within the context of UCM) the causality paths between system responsibilities. Responsibilities are generic system functions (or actions, tasks) that are bound to components, which are abstract organizational structures of the system [3].

A single UCM can contain multiple causality paths between responsibilities and a scenario is a sequence of executions of system responsibilities [7]. A path contains a number of basic path elements.

- Start Point: It is a triggering event and pre-conditions that marks the beginning of a path segment: a UCM has at least one Start Point.
- Responsibility: Represents actions, functions to be performed
- End Point: represents system post-conditions and resulting effects. A UCM can contain more than one end point.

Besides, a number of other path elements (i.e. condition, waiting place, triggers, timer and connection) may exist between a start point and an end point. Furthermore, UCM defines several path connectors to denote concurrent and alternative path such as and-fork, and-join, or-fork, and or-join.

Complex and lengthy scenarios can be decomposed and structured sub map called plug-ins, used in map containers called stubs and displayed as diamonds. Dynamic stubs (dashed diamonds) are used to describe situations where behaviour itself is dynamic.

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 to create and manipulate UCMs that are always syntactically valid. It supports path connections and transformations based on hypergraph-based representations. It also supports the concepts of components and nested levels of stubs and plug-ins. Furthermore, UCM Nav can produce UCMs in XML format that are valid according to Scenario DTD [6]. Figure 3 displays a typical UCMNav interface.

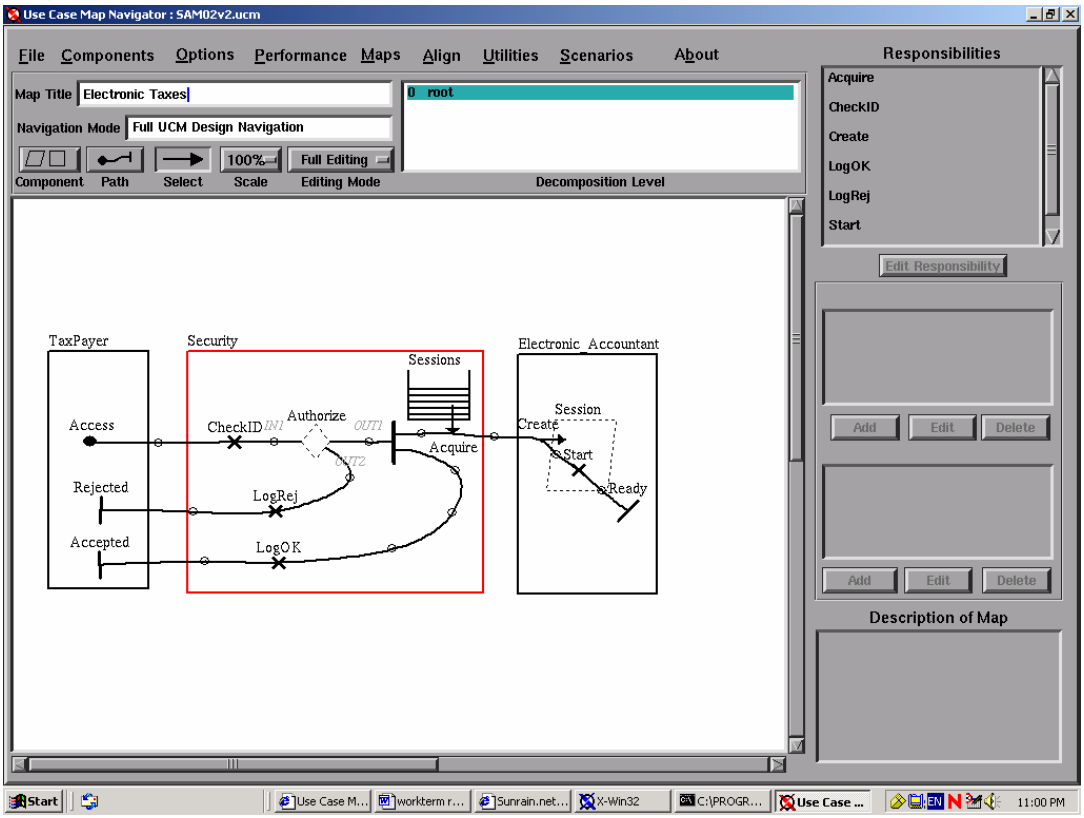


Figure 3: UCMNav Interface

Scenario Definition

UCM Navigator 2.1 can generate scenario definition file in XML using newly developed path traversal algorithm. XML scenario definition files are valid according to scenario DTD. The organization of scenario definition as follows: Scenarios can be parts of a group. A group element can have a set of related scenario elements as its children. Each scenario element can have either one par or seq element. A par or seq element can contain a number of its counterpart as its children. Furthermore, a number of do and condition elements can also be child elements of either par or seq.

This relatively simple representation provides a neutral definition of UCM scenarios. Thus, by implementing various transformation algorithms, it can be used to produce other scenario notations such as MSC, TTCN, UML from UCMs.

Scenario Definition Elements

A single scenario element can represent a path segment or path progression within the original UCM. There are two types of path progression elements: par and seq. The par element is used to describe two or more parallel sequences of responsibilities. A seq element is to define a single sequence of responsibilities. Condition elements capture the conditions satisfied during the traversal of the UCMs (i.e. at choice points and in dynamic stubs). The do elements, which can be of various types such as responsibility, start point, end point, timer, and trigger, are used to model UCM path elements

Message Sequence Chart

Message Sequence Chart (MSC) is the international Telecommunication Union's(ITU) standard language for describing interactions between message-passing instances. MSC is both graphical language and textual notation (Z.120). The graphical representation was produced with a MSC viewer (Telelogic TAU). The two-dimensional diagram give an overview of interaction between communication instance). The textual representation is mainly for data exchange between tools [8].

Transformation

Transformation template rules handle the default mapping of UCM individual scenarios to MSC.

- UCM components become MSC instances
- Responsibilities become action conditions
- Conditions become MSC condition statements
- Timers become MSC timers
- Par elements become MSC par statements
- Start points and end points map to messages coming from or going to the environment
- Connect_start and connect_end (in plug-ins) are dropped

Body

Task 1: Evaluate UML tools for XMI support

The converter is used to transform the XML scenarios into other target languages using XSL Transformation and now transforming the XML scenarios into MSC has been achieved by the converter. Can we extend the work and generate another target scenario representation: UML Sequence Diagrams (represented by XMI) from the scenario definition files. In theory, XML Metadata Interchange (XMI) standard that includes a full XML representation for UML, however, through three weeks research, all known UML tools (Rational Rose, Telelogic TAU, Borland (TogetherSoft), ArgoUML) don't support XMI. The evaluation about UML tools for XMI support is as follows:

1. Rational Rose:
   XMI support is a plug-in that we do not have in our lab. We looked at the Petal file format (.ptl or .mdl), which is also a textual. However, there is no auto-layout for Petal files, so the tool cannot render a graphical sequence diagram correctly
2. Telelogic TAU: supports XMI in theory, but license server problems exists
3. Borland (TogetherSoft): also supports XMI in theory, but license problems exists
4. ArgoUML: supports to export and import XMI, but no auto-layout for imported diagrams, so the tool cannot render a graphical sequence diagram correctly

Task 2: Improve UCM Navigator

Software tool

1. Source-Navigator:

It is a source code analyzing tool. Programmers can edit source code using it, 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. It is especially useful for reverse engineering, since a lot of information, such as class hierarchy, function call relationship, can be extracted from it [5].

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.

The problems with current UCMNav

The current UCMNav can output scenario in XML format. However it can not generate UCM scenario in correct XML format for some Use Case Maps. My Second task in this work term is to improve the UCM Navigator. It consists of four parts:

1. Generate component role attribute for do element
2. Generate component name correctly for responsibility in plug-ins
3. Generate condition element for plug-in selected in a dynamic stub.
4. Generate component id and component name attribute for par element

The solution for above problems

1. Generate component role attribute for do element

A component role attribute related to a component should be given in Scenario XML file. However the component role isn't included in class action which gathers corresponding component information, so I add the data member and associated access methods like setComponentRole and getComponentRole in class action, also put component role attribute into the scenario DTD file. Furthermore, when traversing any type of path data like timer, wait_sych and so on, we need to set related component role as well while setting component name.

2. Generate component name correctly for responsibility in plug-in

Responsibility in a plug-in should have the same component name as the one where the stub is located (unless this responsibility is inside a component in the plug-in), In other words, responsibility in a plug-in should have the name of the component which is the closest to the responsibility. To generate component name correctly for responsibility in plug-in. I design a new traversal scenario algorithm for responsibility. The new traversal scenario algorithm for responsibility is as follows:

- Get a parent map
- Find parent stubs from the parent map
- If installing existing plug-in occurs from parent map, we need to find the stub which encloses that responsibility
- If the stub is enclosed in another stub, we need loop again based on current stub until we find the stub contained in a component.
- Get stub figure from stub pointer which point to current item
- Get containing component name from that stub figure.

3. Generate condition element for plug in selected in a dynamic stub
There are multiple sub-maps when a dynamic stub exists in a UCM map. A condition element and condition attribute associated with each sub map in dynamic stub should be indicated in XML format. Where and when the condition information should be added? The information should be added in traversal scenario algorithm for stub. The algorithm is as follows:
- Get a pluginbinding pointer from sole-plugin by calling SubmapBinding
- Find condition lab from logical-condition which comes from plugin binding
- Then create a condition instance and pass it to functionAddPathTraversal.

4. Generate component id and component name attribute for par element
Although I haven't enough time to fix the problem in this work term, I will discussed it in discuss section.

Implementation
To enhance feature and fix several bugs is mainly accomplished through modifying existing source code such as action.h, action.cc, component.h, conditions.h, path_data.h, resp_ref.cc, result.cc, scenario_generator.cc , scenario_generator.h, start.cc, stub.cc, synchronization.cc, timer.cc, wait.cc, wait_synch.cc and so on

1. In file action.h: declared a char pointer component_role and method void SetComponentRole(const char *) in class Action. In file action.cc: modified constructor of Action class and implement method SetComponentRole by checking whether component role is empty, if no, set new role to component role by calling function strdup, else set component role to NULL. In the same file: also modified method void Action::PrintXMLData(FILE * fp) to save the Action element in XML format
2. In file component_ref.cc: modified method void ComponentReference::SaveXML(FILE *fp, bool ignore_children). In file timer.cc, wait.cc and synchronization.cc: modified method TraverseScenario( ) to set component role.
3. In file start.cc and result.cc: modified method to indicate the name of start point and result point.
4. In file resp_ref.cc, start.cc, and result cc: modified TraverseScenario method to find out enclosed component.

Through testing the modified code on many UCM scenario files, the above problems are solved well.

Task 3: Modify the original XSL transformation algorithm

Environment and tools
The converter development platform is mainly in Win32 environment. It can transform the XML scenario file generated by UCMNav into the target scenario language (i.e.,MSC). The converter contains three main components:

- XML parser: checks the validity of the XML document against the scenario DTD. XSRCES is used in the converter as a validating XML parser that is available at xml.apache.org [9].
- Scenario extractor: extracts individual scenarios from the validate scenario file
- XSLT processor: XALAN is a XSLT processor from Apache to implement the XSLT rules to transform the XML scenario file into another text –based representation. Furthermore, custom rule is used to override the default transformation rules.

Analysis of original XSL transformation algorithm

The original XSL transformation rules provide necessary transformation templates for each XML element of the scenario definition file and different subtypes of a do element. Once each template is finished transforming current element, the next 'following' or 'descendant' element's template is executed. The execution sequence is recursively by the generation of message ids. Every transformation template except 'scenario', receives the next message id as the template parameter. If a message is generated by the current template then the next template will receive an id that is one value greater than the previous one.

However, the algorithm need to be extended because it can't transform some XML files into MSC correctly. There are some bugs to be fixed in the transformation rules. For example, figure 4, 5, 6 present a particular XML file, its corresponding MSC generated by the converter and its MSC generated by UCMNav. Obviously, something is missing in the MSC generated by the converter.

1. Environment instances (or endinstances) need to be generated if the start point and the end point are not bound to a component.
2. The messages coming from or going to the environment need to be generated
3. In parallel, inner-instance messages between components need to be generated Correctly
4. In addition, messages after an AND-Join / MSC par inline statement need to be generated.

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE scenarios SYSTEM "scenarios1.dtd">

<scenarios date ="Tue Feb 18 15:34:39 2003" ucm-file = ".ucm" design-name = "" ucm-design-version = "3">
  <group name = "Group1"  group-id = "1" >
    <scenario name = "MyFirstScenario" scenario-definition-id = "1" >
      <seq>
        <do hyperedge-id="0" name="StartItUp" type="Start" />
        <do hyperedge-id="6" name="Resp1" type="Resp" component-name = "Comp1" component-id = "0" component-role = ""  />
        <do hyperedge-id="8" name="Resp2" type="Resp" component-name = "Comp2" component-id = "1" component-role = ""   />
        <par>
          <seq>
           <do hyperedge-id="20" name="Sin" type="Connect_Start" component-name = "Comp3" component-id = "2" component-role = "" />
           <do hyperedge-id="23" name="Resp4" type="Resp" component-name = "Comp3" component-id = "2" component-role = "" />
           <do hyperedge-id="22" name="Rout" type="Connect_End" component-name = "Comp3" component-id = "2"component-role = ""   />
           <do hyperedge-id="2" name="AllDone" type="End_Point" />
          </seq>
          <seq>
             <do hyperedge-id="17" name="Resp3" type="Resp" component-name = "Comp1" component-id= "0" component-role = "" />
             <do hyperedge-id="16" name="Ongoing" type="End_Point" />
          </seq>
        </par>
      </seq>
    </scenario>
  </group>
</scenarios>
```

Figure 4: a example of an XML file

```
mscdocument MyFirstScenario;
msc MyFirstScenario;

Environment: instance;
Comp2: instance;
Comp3: instance;
Comp1: instance;

text 'scenario: MyFirstScenario';
Environment: out StartItUp,1 to Comp1;
Comp1: in StartItUp,1 from Environment;
            action 'TASK Resp1';
            out m2,2 to Comp2;
Comp2: in m2,2 from Comp1;
           : action 'TASK Resp2';
all: par begin;
Comp2: out m3,3 to Comp3;
Comp3: in m3,3 from Comp2;
            action 'TASK Resp4';
            out AllDone,4 to Environment;
Environment: in AllDone,4 from Comp3;
par;
Comp2: out m5,5 to Comp1;
Comp1: in m5,5 from Comp2;
            action 'TASK Resp3';
            out Ongoing,6 to Environment;
Environment: in Ongoing,6 from Comp1;
par end;
Environment: endinstance;
Comp2: endinstance;
Comp3: endinstance;
Comp1: endinstance;
endmsc;
```

Figure 5: MSC generated by UCMNav

```
mscdocument MyFirstScenario;
msc MyFirstScenario;

Comp2: instance;
Comp3: instance;
Comp1: instance;

text 'scenario: MyFirstScenario';

          action 'TASK Resp1';
          out m1,1 to Comp2;
Comp2:in m1,1 from Comp1;
           action 'TASK Resp2';
all: par begin;
          action 'TASK Resp4';
par;
Comp1: action 'TASK Resp3';
par end;

Comp2: endinstance;
Comp3: endinstance;
Comp1: endinstance;

endmsc;
```

Figure 6: MSC generated by the converter

First, we need to examine the generate_instances or (endinstances) template rule: the template recursively counts the number of remaining descendant do elements whose has component-name as the current 'do' element. If the number equals to zero, then it prints out 'instance' statement. Obviously, a special case is not considered (when descendant do element has no component and its type is “Start” or “End_point”). That is where is the first problem resides.

Second, the write_messages template rule need to be examined: The template print out in and out message statements if next following do element's component name does not match that of current do. The rule is not enough to cover two cases: one is when the current do element has no component name and current do element’ type is “Start” but the next following do element has a component, the other is when the current do element has component name but the next following do element has no component and the following do element’ type is “End_point”. That is where the second problem resides.

Third, also in the write_messages template rule, there is a special case not to be consider (the following element’ s name is “par”) when next following do element's component name does not match that of current do. There is where the third problem resides.

For the fourth problem, because I don’t have enough time in this term, the problem will be fixed later. I still discuss it in discuss section.

Extension of transformation algorithm

According to above analysis, I decide to make some extension for some original XSLT template rules as follows:

1.Generate_instances or (endinstances) template rule:

the template recursively counts the number of remaining descendant do elements whose has component-name as the current 'do' element. If the number equals to zero, then it prints out 'instance' statement. If descendant do element has no component and its type is “Start” or “End_point”, then it print out 'Environment: instance' or 'Environment: endinstance'.

2.Write_messages template rule:

The template print out in and out message statements if next following do element's component name does not match that of current do (however, if the following element’ s name is “par”, it first prints 'all: par begin', then prints out in and out message. If the current do element has no component name and current do element’ type is “Start” but the next following do element has a component, then it prints the messages statement coming from or going to the environment. Also it prints the messages coming from or going to the environment if the current do element has component name but the next following do element has no component and the following do element’ type is “End_point”.

Due to extension of XSLT template rules, the new XSLT template rules is more enough to cover some special cases than original one.

Implementation
The implementation of the extended XSLT template rules is mainly accomplished through modifying generate_instances (endinstances) template rule and write_messages template rule.

First, in generate_instances or (endinstances) template rule, I add a case (when descendant do element has no component and its type is “Start” or “End_point”) to support to generate 'Environment: instance' or 'Environment: endinstance' statements. Then in write_messages template rule, I add two cases (one is when the current do element has no component name and current do element’ type is “Start” but the next following do element has a component, the other is when the current do element has component name but the next following do element has no component and the following do element’ type is “End_point”) to generate messages coming from or going to the environment. Additionally, also in write_messages template rule, I add a special case (when next following do element's component name does not match that of current do element and the following element’ s name is “par”) to fix the third problem.

Through testing the new XSLT code on many scenario XML files, the problems resulted from the original XSLT code are solved well.

Discuss on remaining issues

There are some remaining issues which I haven’t solved in this term.
1. Generate the component name correctly for responsibility in the plug-in map
If the same plug-in map is bound with multiple stubs in one root map, Can we generate the component name correctly for responsibility in the plug-in map? This is a question still not fixed. The reason is that the current traversal algorithm does not work when the same plug-in map is bound with multiple stubs in one root map (discussed in last summer work term). In other words, the current path traversal algorithm can’t determine which stub’s plug-in map is being visited and the responsibility in the plug in don’t know which stub it belongs to, hence the associated component that encloses the stub can not found.

How to solve the problem? I have some suggestion about it. We can know there are two main base classes (Figure and Hyperedge class) and their derived classes (responsibility and responsibilityFigure, stub and stubFigure) involved with fixing the problem through analysis of class hierarchy. However there are only the information of component position (based on component coordinate) but not the information of stub position in these class. In order to know which stub the responsibility in the plug-in belongs to, we need to add a stub position data and associated methods in these classes. The best way I considered to store and retrieve the information of stub position is by using link list. In term of design details, we need further research later.

2. Generate component id and component name attribute for par element
To generate component id and component name attribute for par element, we need to modify associated classes (class action and class synchronization), also add associated component attributes into scenario DTD.

3. Messages generation after an AND-Join / MSC par inline statement.
If there exists a do element after a par element, we need to generate messages after an AND-Join / MSC par inline statement. In order to solve the problem, the write_ template rule need to be extended : if the following do element of a par element exists and the following do element's component name is different from the component names of the last do elements of the par/seq element, it prints in and out messages between associated components.

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 work term is to improve the generation of scenarios from UCM specification

Through four weeks work, with the help of Professor Daniel Amyot, I have completed improving the generation of scenarios from UCM specification. First, I improved the UCMNav to generate UCM scenario in XML format correctly. Then, I modified the converter to implement an extended transformation algorithm. Finally, I provided some suggestion about generating component name correctly for the responsibility in plug-in if the same plug-in map is bound with multiple stubs in one root map. Also, I discussed some remaining issues which I haven't solved like component attribute generation for par element and messages generation after an AND-Join / MSC par inline statement.

During the work term, I learned a lot of new knowledge about UCM and MSC and was familiar with some software-engineering tools: UCM Navigator, Rational Rose and Telelogical TAU. Also, I had a solid understanding of the full software development life cycle. Especially, I developed my work experience in C++, XSLT, software testing, problem solving and OO design and implementation.

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]. Amyot and G. Mussbacher, URN: Towards a New Standard for the Visual Description of Requirements. In: *3rd SDL and MSC Workshop (SAM'02)*, Aberystwyth, U.K., June 2002

[3]. A. Miga, D. Amyot, F. Bordeleau, D. Cameron, and M. Woodside, Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. In: *Tenth SDL Forum (SDL'01)*, Copenhagen, Denmark, June 2001.

[4]. Daniel Amyot, Dae Yong Cho, Xiangyang He, and Yong He, Generating Scenarios from Use Case Map Specifications, Mar 2003

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

[6]. X. He, *Co-op term project*, summer 2002

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

[8]. Dae Yong Cho, Automatic Generation of MSC from UCM Scenario, CSI4900, Fall 2002

[9]. *The XML-Apache Project*, http://www.apache.org/, accessed March 2003.

[10].W3 Consortium, *Extensible Markup Language (XML) 1.0* (Second Edition). W3C Recommendation, 6 October 2000.

[11].W3 Consortium, *Extensible Stylesheet Language Transformation (XSLT) Version 1.0*. W3C Recommendation, 19 November 2000

Appendix

Appendix 1: Modified Scenario Document Type Definition

```
<!--
**************************************************************************
* XML DTD for Use Case Map Scenarios
**************************************************************************
# Authors: Xiangyang He (hexiangyang@hotmail.com)
#         Daniel Amyot (damyot@site.uottawa.ca)
# Version: 1.0
# Organization: SITE, University of Ottawa
# Date: 2002/08/23
# Modified by Shuhua Cui, 2003/04/20
# Root Element: scenarios
-->
<!ELEMENT scenarios (group*)>
<!ATTLIST scenarios
                        date          CDATA    #REQUIRED
                        ucm-file      CDATA    #REQUIRED
                        design-name             CDATA    #IMPLIED
              ucm-design-version      CDATA    #REQUIRED >

<!ELEMENT group (scenario)*>
<!ATTLIST group
               group-id               NMTOKEN            #IMPLIED
               name                   CDATA              #REQUIRED
               description            CDATA              #IMPLIED >

<!ELEMENT scenario (seq | par)>
<!ATTLIST scenario
               scenario-definition-id NMTOKEN            #IMPLIED
               name                   CDATA              #REQUIRED
               description            CDATA              #IMPLIED >

<!ELEMENT seq (do | condition | par)*>

<!ELEMENT par  (do | condition | seq)*>
<!ATTLIST par
                 component-name  CDATA                  #IMPLIED
                 component-role    CDATA                #IMPLIED
                 component-id      NMTOKEN              #IMPLIED >

<!ELEMENT do EMPTY>
<!-- WP_Enter: When the scenario gets to a waiting place
    WP_Leave: After the waiting place is triggered/visited
    Connect_Start: Start point of a plug-in (connection only)
    Connect_End: End point of a plug-in (connection only)
    Trigger_End: End point connected to a start point or waiting place
-->
<!ATTLIST  do
                        hyperedge-id          NMTOKEN                 #REQUIRED
                        name                  CDATA                   #IMPLIED
                         type                 (Resp | Start | End_Point
                                              | WP_Enter | WP_Leave
                                              | Connect_Start | Connect_End
                                              | Trigger_End | Timer_Set
                                              | Timer_Reset | Timeout)      #REQUIRED
                        description           CDATA                   #IMPLIED
                        component-name        CDATA                   #IMPLIED
                        component-role        CDATA                   #IMPLIED
                        component-id          NMTOKEN                 #IMPLIED >
<!ELEMENT condition EMPTY>
<!-- expression is the boolean expression used in the selected branch -->
<!-- label is the name associated to the next empty point in that branch -->
<!ATTLIST condition
                        hyperedge-id          NMTOKEN #REQUIRED
                        label                 CDATA    #REQUIRED
                        expression            CDATA    #IMPLIED >
```

Appendix 2: An scenario output after the UCMNav is modified

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE scenarios SYSTEM "scenarios1.dtd">

<scenarios date ="Thu  Mar  27 13:31:19 2002" ucm-file = "SAM02.ucm" design-name = "SAM02" ucm-design-version = " 43">
  <group name = "Biometrics"  group-id = "1" >
    <scenario name = "BioOK" scenario-definition-id = "1" >
      <seq>
        <do hyperedge-id="0" name="Access" type="Start" component-name = "TaxPayer" component-id= "5" component-role=
""/>
        <do hyperedge-id="3" name="CheckID" type="Resp" component-name = "Security" component-id= "2" component-role=
""/>
         <condition hyperedge-id="24" label="Biometrics" expression ="(!bv3)&amp;amp;(!bv4)" />
        <do hyperedge-id="25" name="Bio" type="Connect_Start" component-name = "Security" component-id= "2" component-
role= ""/>
        <do hyperedge-id="32" name="GetBio" type="Resp" component-name = "Security" component-id= "2" component-role=
""/>
        <do hyperedge-id="27" name="CheckBio" type="Resp" component-name = "BioDB" component-id= "0" component-
role= "" />
        <condition hyperedge-id="28" label="[BioOK]" expression ="bv1" />
        <do hyperedge-id="35" name="Continue" type="Resp"  component-name = "Security"  component-id= "2" component-
role= "" />
        <do hyperedge-id="26" name="Yes" type="Connect_End" component-name = "Security" component-id= "2" component-
role= ""/>
       <par>
          <seq>
            <do hyperedge-id="17" name="Acquire" type="Resp" component-name = "Security" component-id= "2" component-
role= "" />
            <do hyperedge-id="19" name="Create" type="Resp" component-name = "Electronic_Accountant" component-id= "1"
component-role= "" />
            <do hyperedge-id="21" name="Start" type="Resp" component-name = "Session" component-id= "3" component-role= ""
/>
            <do hyperedge-id="2" name="Ready" type="End_Point" component-name = "Electronic_Accountant" component-id= "1"
component-role= ""/>
          </seq>
          <seq>
            <do hyperedge-id="11" name="LogOK" type="Resp" component-name = "Security" component-id= "2" component-
role= "" />
            <do hyperedge-id="8" name="Accepted" type="End_Point" component-name = "TaxPayer" component-id= "5"
component-role= ""/>
          </seq>
        </par>
      </seq>
    </scenario>
  </group>
</scenarios>
```