Project Report

Automatic Generation of MSC from UCM Scenario

Name: Dae Yong Cho

ID: 1338984

Supervisor: Daniel Amyot

TABLE OF CONTENTS

1. ABSTRACT

The UCM provides a high-level view of functional requirements and the causality flow between different elements of proposed system. The UCM is used to model both structural and behavior aspects of the system. Furthermore, the UCM permits the client to define a set of Boolean variables that are used in UCM path traversal. A path generated by the traversal is called scenario. The UCMNav is a software-engineering tool that is capable of producing a scenario definition file, which includes a set of scenarios categorized into groups. The main goal of the Converter is to produce basic MSCs from the scenario definition file.

2. INTRODUCTION

The Converter is a tool that produces message sequence charts (MSC) in textual Z.120 format from **UCMNav** generated scenario file and its original use case map (UCM) file. A scenario constitutes a path segment within the context of the UCM (the UCM might be a concurrent model of a set of scenarios in which case a set of related scenarios are generated by **UCMNav**). A scenario always has a start point and an end point. In between those two elements a number of other path elements (i.e., responsibility, condition, waiting place, trigger, timer, and connection) may exist. The scenario file assembles a set of related scenarios into groups. The goal of the tool is to produce a basic MSC of a single scenario.

UCMNav supports the notion of scenario definitions. The feature, through global variables for path selection, allows the user visually highlight a single scenario and used to produce other representation of scenarios like Message Sequence Charts. Initially path traversal and generation of MSCs are combined into one algorithm. However, due to its complexity, the resulted code was difficult to maintain and it often generated incorrect path traversals.

To remedy the situation, UCMNav 2.0 de-couples the UCM path traversal algorithm from MSC generation. The tool now generates neutral scenario definition in XML. The main goal of the project is to produce a tool that can automatically generate MSCs from UCM scenario definition files and investigate tools and infrastructure that we can use to produce other types of scenario representations.

3. USE CASE MAP

Description

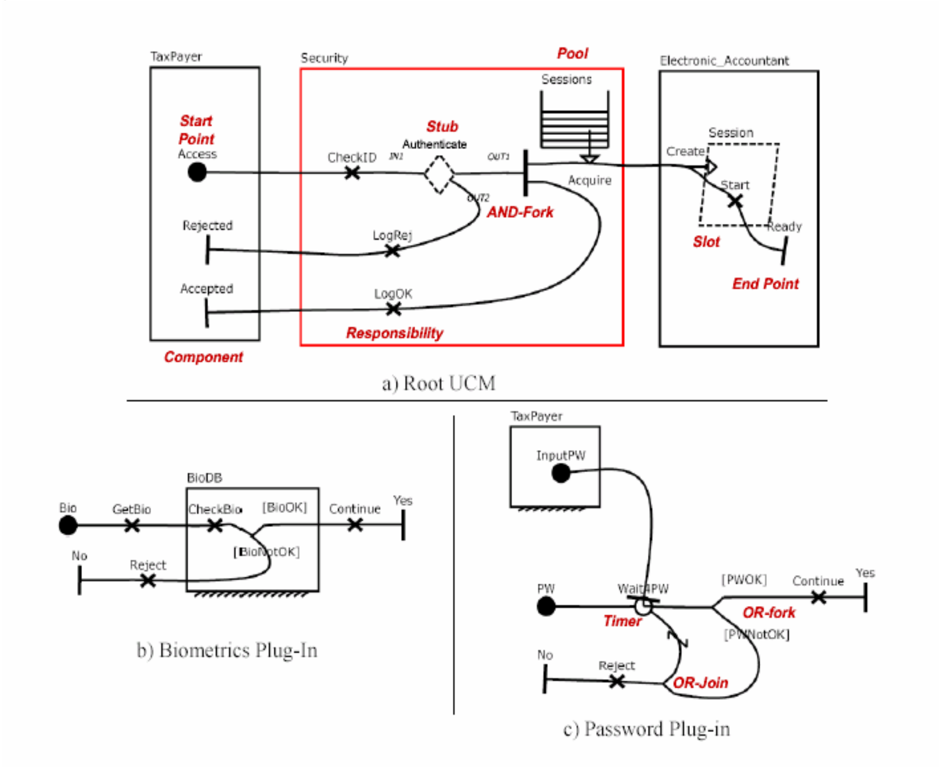


Figure 1: Examples of Use Case Map

Use Case Map (UCM) is a scenario-based notation for modeling functional requirements. It is one of the two modeling techniques that constitute User Requirement Notation (URN) standard [X]. The main intent of the notation is to provide a way to express and validate high-level architectural designs of concurrent real-time systems at very early stages of the software development process.

Use Case Map's approach for describing systems behaviors is based on the scenarios and the scenarios are (within the context of UCM) the causality flows between system responsibilities. Responsibilities are generic system functions (or actions) that are (optionally) bound to components, which are abstract organizational structures of the system [sdl01].

UCM Scenarios

A single UCM can contain multiple causality paths between responsibilities and each deterministic path is referred to as a scenario. In other words, a scenario is a sequence of executions of system responsibilities. A path contains a number of basic path elements [sam2000].

They are…

- **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**. Responsibilities are high-level system elements that are executed as the responses to a triggering event.
- **End Point**. An End Point is defined as system post-conditions and after effects. Like Start Point, a UCM can contain more than one End Point.

Responsibilities can be superimposed on top of Components. Components are system-level organizational structure abstraction. Furthermore, UCM defines several path connectors such as AND-fork, AND-join, OR-fork, and OR-join for denoting concurrent path progressions and alternatives [sam2000].

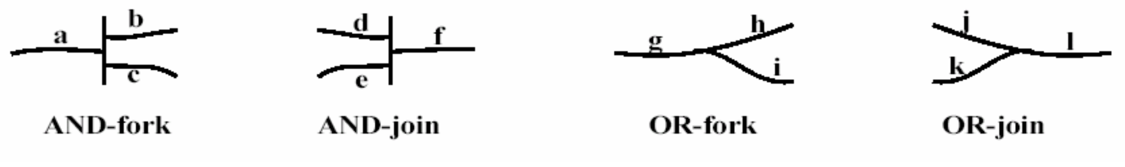
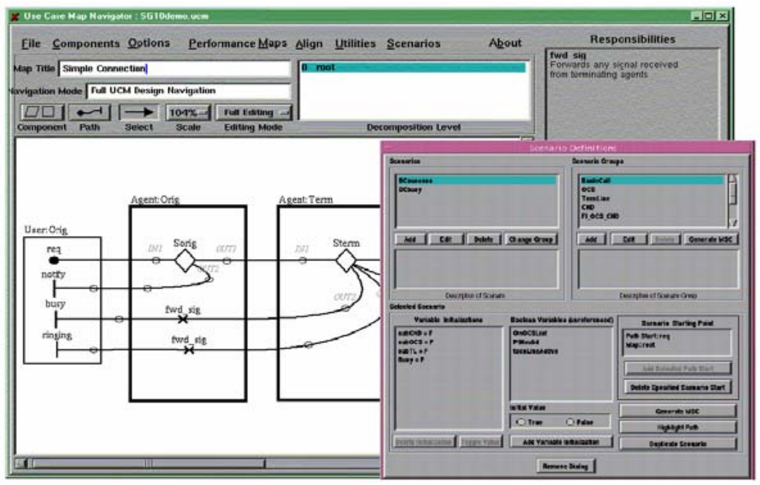


Figure 2: Path connectors

UCM Navigator

UCM Navigator is a platform neutral software engineering tool for UCM notation. It is mainly work of Andrew Miga at Carleton University. The tool allows the user to visually create and manipulate UCMs that are always syntactically valid. It supports path connections and transformations based on hypergraph-based semantics [web]. Also, it fully supports the concepts of components and nested levels of stubs and plug-ins.



Map Title: Simple Connection
Navigation Mode: Full UCM Design Navigation
Scale: 104%
Editing Mode: Full Editing
Decomposition Level: 0 root

fwd_sig
Forwards any signal received from terminating agents

Scenarios: BCsuccess, BCbusy
Scenario Groups: BasicCall, OCS, TermLine, CND, FI_OCS_CND
Variable Initializations: subCND = F, subOCS = F, subTL = F, Busy = F
Boolean Variables (unreferenced): OnOCSList, PMode, [illegible]
Scenario Starting Point: Path Start: req, Map: root

Figure 3: UCMNav Interface

Moreover, UCMNav can output UCMs in XML that are valid according to UCM document type definition (DTD).

Scenario Definition

UCM Navigator 2.1 incorporates the notion of scenario definition and can generate scenario definition file in XML using newly developed path traversal algorithm [work-term]. XML scenario definition files are valid against scenario DTD. The organization of scenario definition as follows: a set of related scenario elements are gathered as child elements of a group element. Each scenario element can have either one par or seq element. Recursively, 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.

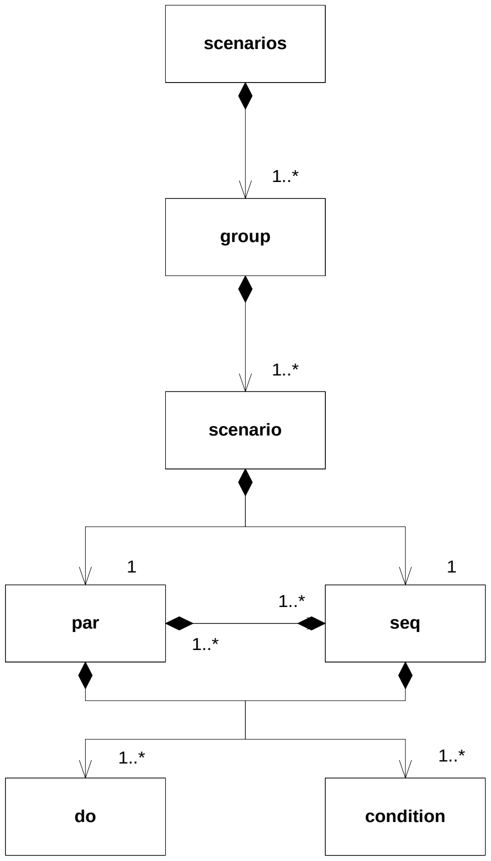


Figure 4: The tree view of scenario document type definition

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

Scenario Definition Elements

Logically, a single scenario element can represent a path segment or path progression within the original UCM. As the intent of UCM is to model concurrent system behavior, two types of path progression elements, par and seq, are mapped into scenario DTD. The par element is used to describe two or more parallel sequences of responsibilities and as previously stated it can also contain a number of do and condition elements as its direct child nodes. Where as, seq element is to define a single sequence of responsibilities.

Condition elements are used to denote both pre and post conditions of path connectors (par, seq) and path elements (do). The do elements are used to model UCM path elements such as Responsibility, Start point, End point, Timer, and Trigger.

4. MESSAGE SEQUENCE CHART

Description

Message Sequence Chart (MSC) is the International Telecommunication Union's (ITU) standard language for describing interactions between message-passing instances. Its purpose and functions are similar to UML sequence diagram at basic level. MSC is both graphical language and textual notation (Z.120). The two-dimensional diagrams give an overview of interactions between communicating instances. The textual form of the language is mainly for data exchange between tools.

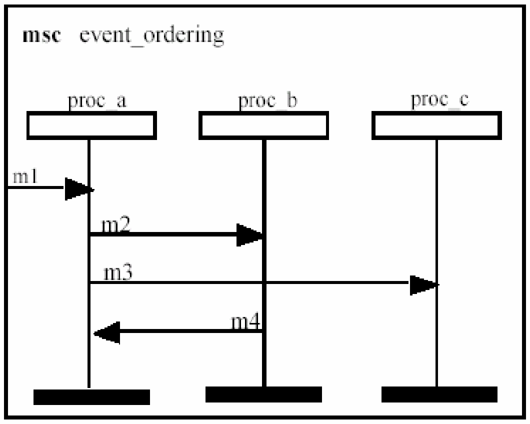


Figure 5: Basic MSC

A global clock is assumed by one Message Sequence Chart [z.120]. Each timeline below each instance indicates its lifetime. However, no time scale is given. This means that MSC provides an ordered view of events such as inter-component messages and actions, but does not provide actual duration of each event. Like UCM, MSC also support concurrent sequences of events.

Following is the list of some of notations supported by MSC.

- **in**: This statement models in going message from an instance to another.
- **out**: 'out' statement represent out going message from an instance to another.
- **instance**: 'instance' statement marks the beginning of the lifeline of an instance or component of the system.
- **endinstance**: 'endinstance' statement signifies the end of an instance.
- **action**: the 'action' denotes an action or method performed by an instance.
- **set**: 'set' is used to mark a timer start.

- **reset**: 'reset' is used to mark a timer end.
- **timeout**: This statement represents the timer timeout.
- **All: par begin**: The 'All: par begin' statement marks the starting of parallel processing.
- **end**: This statement represents the end of parallel processing.

```
mscdocument BioOK;
msc BioOK;

BioDB: instance;
Session: instance;
Electronic_Accountant: instance;
Security: instance;
TaxPayer: instance;

text 'scenario: BioOK';

TaxPayer: out Access,1 to Security;
Security: in Access,1 from TaxPayer;
          action 'CheckID';
          action 'GetBio';
          out m2,2 to BioDB;
BioDB: in m2,2 from Security;
       action 'CheckBio';
       condition [BioOK];
       out m3,3 to Security;
Security: in m3,3 from BioDB;
          action 'Continue';
all: par begin;
Security: action 'Acquire';
Security: out m4,4 to Electronic_Accountant;
Electronic_Accountant: in m4,4 from Security;
                       action 'Create';
                       out m5,5 to Session;
Session: in m5,5 from Electronic_Accountant;
         action 'Start';
         out Ready,6 to Electronic_Accountant;
Electronic_Accountant: in Ready,6 from Session;
par;
Security: action 'LogOK';
Security: out Accepted,7 to TaxPayer;
TaxPayer: in Accepted,7 from Security;
end;

BioDB: endinstance;
Session: endinstance;
Electronic_Accountant: endinstance;
Security: endinstance;
TaxPayer: endinstance;

endmsc;
```

Figure 6: An example of MSC (BioOK)

5. TRANSFORMATION

Relationship between UCM and MSC

In order to achieve proper transformation of UCM scenario to basic MSC, we must first clearly identify the relationship between two scenario modeling techniques. Both UCM and MSC focus on the scenario description. However, there are few major differences in their respective concepts. Here is a list of their differences that are relevant to the transformation.

- **Level of Scenario Abstraction**. The first dissimilarity is the abstraction level at which each notation models scenarios. In UCM, scenarios are modeled as sequences of responsibilities and inter-component interactions are abstracted away. By contrast, (basic) MSC describes scenarios as a sequence of inter-instance message, and action.
- **Component and Instance**. Within the context of UCM, a component is a role-play by scenario and it has a set of responsibilities and it is optional. On the other hand, MSC instances express separate location and they are mandatory.
- **Condition**. When a condition is modeled into a UCM scenario, it is explicitly defined as a pre or post condition for other UCM elements. However, MSC doesn't make such distinction and conditions are used to describe System State.

Relationship between elements of UCM scenario and MSC

Besides identifying conceptual differences, to develop transformation algorithm we also have to map one-to-one relationship between elements of UCM scenario definition file and basic MSC.

- UCM component to MSC instance.
- scenario element to basic MSC.
- par element to MSC par statement.
- condition element to MSC condition statement.
- do elements to MSC messages, actions, and timers.

6. Extensible Style Language

Description

The e**X**tensible **S**tyle **L**anguage is a W3C standard that describes how to define XML stylesheets. The stylesheet provides exact description on how to represent the structured content of a XML document in other textual notations such as HTML, SVG, and VRML. In other word, it shows how to lay out and style source XML document onto some other presentation medium.

An XSL processor receives a XML document and XSL stylesheet as inputs and constructs a presentation of source XML document as intended by the author of the stylesheet. The process is divided into two phases: XSL Transformation and XSL Formatting. The XSL Transformation refers to the result tree construction phase and the XSL Formatting represents the process of formatting the result tree into the target presentation medium and it is achieved by the formatter (the rendering engine inside of web browser can be a formatter).

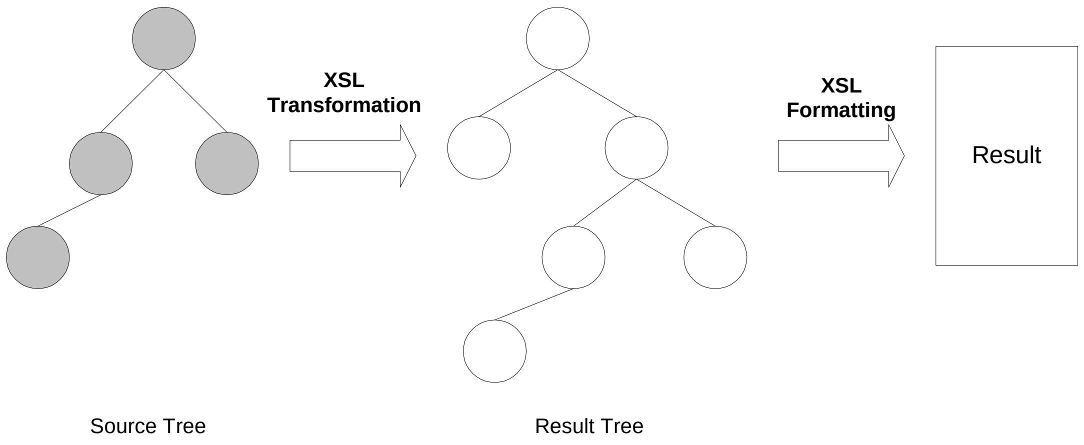


Figure 7: XSL Process

XSLT

As mentioned before, the XSL Transformation refers to the part of XSL that allows the construction of the result tree. Within the context of XSL, the result tree is called the **element and attribute tree**. In order to construct the result tree, the construction rules embedded in the stylesheet are used. Each tree construction rule is consisted of a pattern that is matched against element(s) in source XML document and a template that contains precise instructions on how to construct the corresponding portion of the result tree. This means that the same stylesheet can be applied to XML documents that have similar structure (i.e., same DTD).

The XSL Stylesheet itself is a XML document and it contains a set of result tree construction rules (i.e., templates). The rules can specify literal result element structure or can be XSLT elements that are used to create result tree fragments. The XSLT processor such as Xalan scans through the source tree and instantiates appropriate templates. When a template is invoked it is always respect to the current node, which is a member current node list.

There are two types of transformation templates within XSLT. The first is 'named' template (i.e., `<xsl:template name = "xxx" >`). A named template can be explicitly called within the XSL file. In most cases, it is used to encapsulate repetitive transformations that are not specifically tied to a XML pattern. The second template type is 'matched' template (i.e., `<xsl:template match = "xxx" >`). This kind of templates are called by the XSLT processor when it encounter a XML element after the `<xsl:apply-templates select = "xxx" />` statement.

XPath

The XPath is a part of XSLT whose main purpose to address a portion of XML document. In addition, it is also used to for pattern matching. The basic XPath syntax is similar to the file system addressing. For example, if the path expression starts with a slash ('/'), then it indicates an absolute path. XPath also defines a library of standard functions for working with strings, numbers and Boolean expressions.

Here are examples of XPath notations.

- **ancestor**: ancestor represent all nodes that is parent of current node or parent of parent of current node and so on so forth.
- **descendant**: descendant represent all nodes that is child of current node of child of child of current node and so on so forth.
- **parent**: the parent of current node.
- **child**: any child of current node.
- **following**: any node after the current node excluding any descendants.
- **following-sibling**: any sibling node that is after the current node.
- **preceding**: any node that precedes the current node excluding any child node of its preceding siblings.
- **preceding-sibling**: any sibling node that precedes the current node.

7. IMPLEMENTATION

ENVIRONMENT & TOOLS

Compiler

The initial software development and testing will proceed in Win32 environment. The program will be first developed as a Win32 console application using VisualC++ compiler. At the same time all platform specific code will be identified and marked. Once program logic and structure have been verified and tested, the source code will then be ported to Cygwin and will be re-compiled using GNU C++ compiler (g++) provided in the Cygwin environment.

XML Parser

The Xerces is a XML parser that is freely available at xml.apache.org. The parser, which is initially developed by IBM, confirms to XML 1.0 recommendation and supports both SAX and

DOM APIs. Contrast to Expat, Xerces is a validating parser, which means it checks validity of the XML document against constraints described in the DTD as well as well-formedness of the document. This is important in our project since all scenario files must conform to the scenario file DTD to be valid. The source code of the parser is platform-independent and various binary ports to different platforms are available including Win32, Linux, and Unix.

Unlike Expat, which only supports SAX and event-driven programming approach, the Xerces supports Document Object Model (DOM) API in addition to SAX. DOM creates tree-based in memory representation of the XML document. This approach to XML document might prove to be useful when generating MSCs from scenario file since it eliminates the needs to parse the document for a specific scenario (at the expense of memory).

XSLT Processor

The Xalan is a XSLT processor that is also freely available at xml.apach.org. Xalan is a free XSLT processor from Apache that implements the XSL Transformations (XSLT) Version 1.0 and the XML Path Language (XPath) Version 1.0. It allows API programmers to transform an XML document into another text-based representation such as HTML. [**note**: currently, Xalan does not supports cygwin and I was unable to compile Xalan under cygwin.]

TRANSFORMATION ALGORITHM

The Algorithm

The scenario XSL file contains all transformation templates necessary for deriving basic MSC representation (in textual Z.120) of a scenario from the given UCM scenario definition file.
The overall algorithm is relatively simple. Each XML element of the scenario definition file has a corresponding 'matched' template (i.e., `<xsl: template match = "xxx"`) and different subtypes of a 'do' element also have corresponding named template (i.e., `<xsl: template name = "xxx"`). Once each template is done transforming current element, the execution control is passed on to the next '`following`' or '`descendant`' element's template. The execution sequence is almost recursive in nature and it is due to 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.

Scenario

```
<xsl:template match = "//scenario">
<xsl:param name = "msgid" />
…
</xsl:template>
```

Figure 9: scenario matched template

The 'scenario' template marks the beginning of scenario transformation. When the transformation algorithm detects the presence 'scenario' XML element, it invokes the 'scenario' matched template. The template first prints out the MSC document header follows by 'instance' statements and the scenario header. Then it applies the matched template for the first direct child element of the 'scenario'. When the execution control returns to the template, it generates 'endinstance' statements and writes the document footer.

Instances

```
<xsl:template name = "generate_instances">
…
</xsl:template>


<xsl:template name = "generate_endinstances">
…
</xsl:template>
```

Figure 10: generate_instances and generate_endinstance named templates

The Z.120 dictates that the basic MSC begins with '`intance`' statements and ends with '`endinstance`' statements. Both '`instance`' and '`endinstance`' identify unique instance names used in the basic MSC. Both algorithms that generate above statements from UCM scenario are almost identical. The algorithms sequentially process '`descendant`' '`do`' elements one by one and count the number of remaining '`do`' element with the same '`component-name`' attribute. When there are no more remaining '`do`' with the same component name, they generates an '`instance`' or '`endinstance`' statement.

Inter-instance Messages

```
<xsl:template name = "generate_message">
<xsl:param name = "msgid" />
…
</xsl:template>
```

Figure 11: generate_message named template

The most complex of part of the transformation is the generation of messages between instances. When two '`do`' elements in sequence have different component-name attribute this represents the UCM path crossing from one component another. In such case, the transformation algorithm should generate a MSC inter-instance message (i.e., 'in' message and '`out`' message statement pair). However, there is a case when the generation message is unwarranted. When current '`do`' element is the last child element of '`seq`'. The first example is the '`do`' element that has the '`End_Point`' as the type attribute. This element signifies an end point of the scenario (scenario may have multiple end points) thus no message should be generated - even the following '`do`' element has different component-name attribute. The second example is the last child 'do' element of '`seq`' that is again one of the (direct) child element of '`par`'. Such set of '`seq`' elements models parallel executions of system responsibilities. And since there could be a second '`seq`' element following the parent (i.e., `seq`) of the current '`do`', the algorithm must not produce messages between the current 'do' and its '`following`' 'do' element. However, above two cases somewhat intersect. If the scenario has multiple end points, then they may be encapsulated in the '`par/seq`' as the last 'do' element of each '`seq`' child element of the '`par`'.

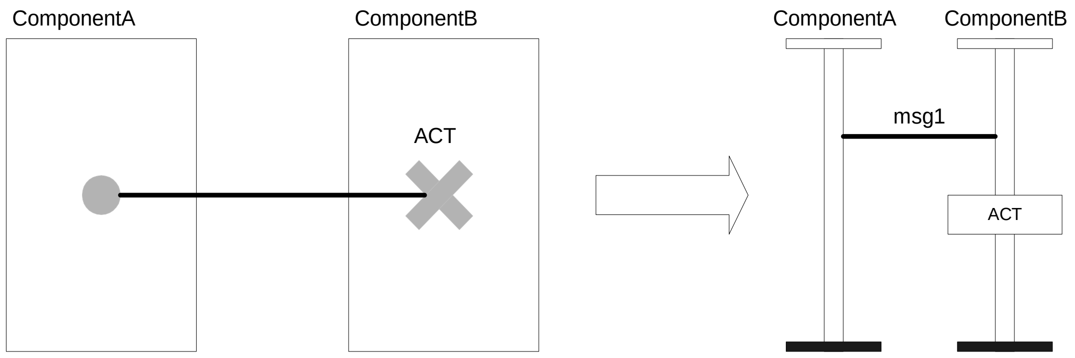


Figure 12: MSC message

Each inter-instance MSC message must have a unique id. Because, messages are derived, the transformation algorithm itself must tag each message with a unique number. The problem is XSLT does not provide adequate mechanisms, which one can use to generate a unique number. Furthermore, the lack of variable construct (yes XSL has '`XSL:variable`' but it is more like a constant than a variable since its value can not be reset) further compounds the problem. This means that as the algorithm process each XML element it must pass the next message id to the transformation template of next element as its parameter (i.e., `xsl:param`). If an inter-instance message is generated, then the text template receives the id that is one value greater than the previous id.

In most cases, the messages that are generated by the transformation algorithm have generic names (i.e., m_x). However, there are two instances when the value of do's name attribute is used instead. The first case is the '`Start`' 'do' element. When the algorithm encounters the '`Start`' 'do' element, it uses the name of 'do' element as the message name. The second case is when '`following`' 'do' element's (of current 'do') type attribute is '`End_Point`'. In such as the name of '`End_Point`' 'do' is used.

Parallel Sequences

```
<xsl:template matched = "par" >
<xsl:param name = "msgid" />
…
</xsl:template>


<xsl:template matched = "seq" >
<xsl:param name = "msgid" />
```

```
…
</xsl:template>
```

Figure 13: par and seq matched templates

Like UCM, MSC supports parallel sequences of events. The beginning of parallel sequence is marked with 'all: par;' statement and the end with '`end;`'. Each parallel sequence in between those two statements begins with '`par;`'. When the algorithm process the '`par`' template, the template prints out '`all: par;`' but not "`end;`'. This is because the algorithm first processes all 'following' and 'descendant' elements before returning to the par template. Thus, printing out '`end;`' is unadvisable. Instead an 'end;' is generated when the algorithm processes the last do element of the last child '`seq`' element of par. Similarly, the '`seq`' element could be stand-alone or a child element of a '`par`'. If the current '`seq`' is a child of '`par`' and it has '`preceding-sibling`' that is again '`seq`', then we print out '`par;`' statement to mark the beginning of the new parallel sequence.

Actions

```
<xsl:template name = "Resp" >
<xsl:param name = "msgid" />
…
</xsl:template>
```

Figure 14: Resp named template

MSC 'action' statement represents an action (or responsibility) that is taken by a specific instance. When the algorithm encounters a 'do' that has '`Resp`' as its type attribute, it outputs '`action`' follow by the value of 'component-name' attribute.

Conditions

```
<xsl:template match = "condition" >
<xsl:param name = "msgid" />
…
</xsl:template>
```

Figure 15: Condition matched template

MSC also support conditions, but unlike UCM it does not distinguish between post and pre conditions. When the transformation algorithm processes a do element, before proceeding to next element or generating inter-instance message, it writes out all '`following-sibling`' conditions of the current '`do`'. This may contradict the intention of the scenario since the conditions may be pre-conditions of path branching (i.e., 'par') or the following 'do' element. However, the end product is the same for MSC.

Timer

```
<xsl:template name = "Timer_Reset">
<xsl:param name = "msgid" />
…
</xsl:template>


<xsl:template name = "Timer_Set">
<xsl:param name = "msgid" />
…
</xsl:template>


<xsl:template name = "Timeout">
<xsl:param name = "msgid" />
…
</xsl:template>
```

Figure 16: Timer named templates

Both UCM scenario and basic MSC implements the timer functions and the notations used in both languages are relatively similar. Within UCM scenario definition, behaviors of timer are encoded as do element's type attribute and element's name attribute is the name of the timer. When the algorithm sees a 'do' with 'Timer_Set' as the value of type attribute, it first prints out 'action' statement for the owner component of 'do' to indicate that the component is waiting for a timer. Then, it outputs 'set' with the value of the 'do' element's name attribute to begin the timer. Similarly, for Timer_Reset type attribute, it again writes the 'set' statement. Finally, the 'Timeout' type attribute of 'do' perform literal translation to 'timeout' in basic MSC.

FUTURE DESIGN

Message ID and Overall Design

It is author's opinion that the design of the overall transformation algorithm is relatively straightforward but it is also a bit difficult to maintain and to extend due to its 'recursive' nature. With the current design, it is impossible to mark the end of an element (ex: 'All: par begin' and 'end;') within its template because all following and descendant elements are processed before the execution control returns to the current element's template. A better design could have achieved if the scenario template applies templates for each its descendant elements (ex: '<xsl: for-each select = "xxx">') and the hyperedge-id attribute of do element is used as the message id rather than generating a unique number.

Extending Templates

The XSLT provides two mechanisms to extend or change behaviors of transformation templates. One could add more transformation templates by including '`<xsl:include href = "xxx.xsl" />`' statements. This statement enables the XSLT processor to seek out missing templates from the specified XSL file. Other mechanism is the '`<xsl:import href = "xxx.xsl" />`'. Unlike the 'include' statement, this tells the processor to replace original template with the template that is embedded in the imported XSL file. Furthermore, one can add extra transformation behavior by inserting '`<xsl:import href = "xxx.xsl" />`' statement to the template. This statement directs the XSLT processor to insert the transformation behavior of the template in the imported file where the statement is inserted. Unfortunately, Xalan only supports insertion of second template behavior and do not permit to replace a whole template.

8. CONCLUSION

As XML is platform and application independent, by encoding the UCM scenarios in XML, we were able to separate scenario generation from its representation. Also, it offers a way to performed continued analysis of the high level system design using other notations such as MSC and UML sequence diagram. The E**x**tensible **S**tylesheet **L**anguage Transformation provides the best generic solution for transforming the scenario definition files to another textual representation of scenario such as basic MSC. Since it is XML based, its very flexible and changing transformation behavior does not require re-compilation of the application. Furthermore, the transformation behavior encoded in the original XSL file could be easily extended using the extension mechanisms provided by the XSLT.

However, there are more works that need to be done. In order to achieve proper transformation of the given scenario without referring to the original UCM file, all necessary information has to be presented in the XML scenario definition file. Moreover, the scenario definition has to be general enough to be transformed into another languages than MSC.

9. REFERENCES

1. F. Bordeleau, D. Cameron, On the Relationship between Use Case Maps and Message Sequence Charts, [on-line], Available: http://www.usecasemaps.org/pub/sam2000.pdf , 2000

2. XiangYang He**,** Path Traversal and Scenario Generation from Use Case Map, 2002

3. Andrew Miga, Daniel Amyot, Francis Bordeleau, Donald Cameron, Murray Woodside, Deriving Message Sequence Charts from Use Case Maps Specifications, [on-line], Available:
http://micmac.mitel.com/conferences/2001/presentations/Andrew_Miga_MICON_Presentation.pdf
,2001

4. Daniel Amyot, Gunter Mussbacher, URN: Towards a New Standard for the Visual Description of Requirements, [on-line], Available: http://www.usecasemaps.org/pub/sam02-URN.pdf, 2002

4. Nic Miloslav, XSL Tutorial, [on-line]
Available: http://www.zvon.org/HTMLonly/XSLTutorial/Books/Book1/

5. XSL Transformations (XSLT) Version 1.0, [on-line] Available: **http://www.w3.org/TR/xslt** , 2001

6. Message Sequence Chart (MSC) Z.120, [on-line], Available:
http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-Z.120-199911-I, 1999