Generating Scenarios from Use Case Map Specifications

Daniel Amyot, Xiangyang He, and Yong He
SITE, University of Ottawa
800 King Edward
Ottawa, Ontario, Canada, K1N 6N5
{damyot | u2417281 | yonghe}@site.uottawa.ca

Dae Yong Cho
Samsung Electronics Corp. Tech. Operations,
Software Center, Apgoojung building, 5th floor
Sinsa-dong 599-4, Kangnam-goo, Seoul, Korea
daeyong.cho@samsung.com

Abstract

The Use Case Map (UCM) notation is being standardized as part of the User Requirements Notation (URN), the most recent addition to ITU-T's family of languages. UCM models describe functional requirements and high-level designs with causal paths superimposed on a structure of components. The generation of individual scenarios from UCM specifications enables the validation of requirements and facilitates the transition from requirements to design. In this paper, we address the challenges faced during the automated generation of such scenarios. Scenario definitions and traversal algorithms are first used to extract individual scenarios from UCMs and to store them as XML files. Transformations to other scenario languages (for instance, Message Sequence Charts) are then achieved using XSLT. Possible applications of this two-step generation process include early validation and synthesis of design models. Illustrative examples are given based on our current tools and recent experiments.

Keywords: Message Sequence Charts, Scenarios, Transformations, Use Case Maps, Validation, XML

1. Introduction

For more than a decade, scenarios have proven to be effective tools for designing, validating, and thinking about complex software found in various applications, including reactive and distributed systems. Scenarios are often used to capture and communicate functional requirements, but they also represent a solid basis for testing and maintenance activities. Several scenario notations widely accepted in practice include Message Sequence Charts (MSCs) [8] and UML use cases, sequence diagrams, and activity diagrams [15].

The *Use Case Map* notation, one of the latest additions to the ITU-T family of languages, is also scenario-oriented. It is being standardized as part of the User Requirements Notation in the Z.150 series of Recommendations [1][9]. UCM graphical models describe functional requirements and high-level designs with causally linked responsibilities, superimposed on structures of components [3][10]. UCM *responsibilities* are scenario activities representing something to be performed (operation, action, task, function, etc.). A responsibility can potentially be associated or allocated to a *component*. In UCMs, a component is generic and abstract enough to represent software entities (e.g. objects, processes, databases, or servers) as well as non-software entities (e.g. actors or hardware).

UCMs are useful for describing multiple scenarios abstractly in a single, integrated view. This promotes the understanding and reasoning about the system as a whole, as well as the early detection of conflicting or inconsistent scenarios [2]. However, for complex UCMs (e.g. with many alternative paths or with multiple levels of submaps), individual scenarios become difficult to recover. Yet, individual scenarios contribute greatly to the understanding of particular functionalities, and they also guide the definition of more detailed scenarios for the design phase, for validation test cases, and for documentation. Hence, it is important to be able to select or extract individual scenarios from a complex UCM requirements specification.

The extraction of scenarios from UCMs is not without challenges. UCMs are very abstract in nature, and they can be constructed and combined in very flexible ways. This problem was first tackled by Miga *et al.* in [14], and their solution was prototyped in the UCM Navigator (*UCMNAV*, [13]), a multi-platform tool written in C++. This tool can highlight the UCM paths traversed according to scenario definitions (to be discussed in Section 3), and generate individual scenarios in the form of Message Sequence Charts using the Z.120 textual syntax [8]. However, their approach is very limited and inflexible because the traversal of UCMs is intertwined with the generation of the target scenario representation (which makes the programming error-prone and difficult to maintain), and there is no way to adapt the traversal algorithm or modify the transformation to a different target scenario language.

In this paper, we present a novel two-step approach to the generation of scenarios from UCMs that is robust, flexible, and extensible. Our approach decouples the traversal of UCMs from the generation of the target scenarios, and hence reduces the complexity of the algorithms and programs involved. The result of such traversal can be stored in an XML file [17], whose structure is presented in Section 2.4. The XML files become the basis for a second step, which consists of transforming the XML scenarios into scenarios in a specified target language (e.g. MSCs or UML sequence diagrams). We are using *XSL Transformations* (XSLT, [18]) because XSLT is well suited for XML-based transformations to various text-based representations, and 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 (e.g. specific messages). Many XSL specifications can be used on the same XML scenario file to provide transformations to various target languages. Figure 1 gives an overview of this scenario generation process, to be detailed in the next sections.

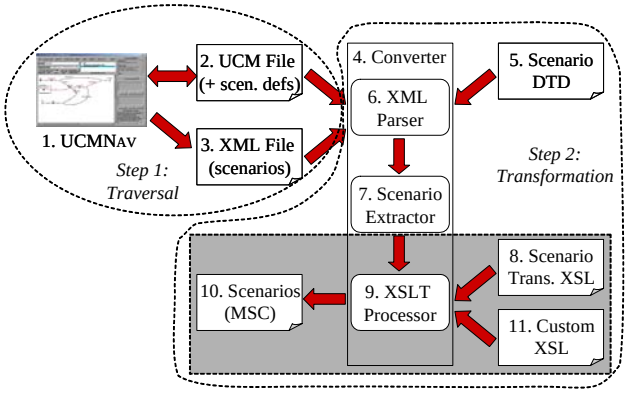


Figure 1 Overview of the scenario generation process

The rest of the paper is organized as follows. Several challenges in the generation of scenarios from UCMs are presented in Section 2, together with an overview of our solutions. The scenario Document Type Definition (DTD) and traversal algorithms are then discussed (Section 3). An example XSLT transformation to MSCs is described in Section 4. A brief discussion is found in Section 5, followed by our conclusions.

2. Scenario generation challenges

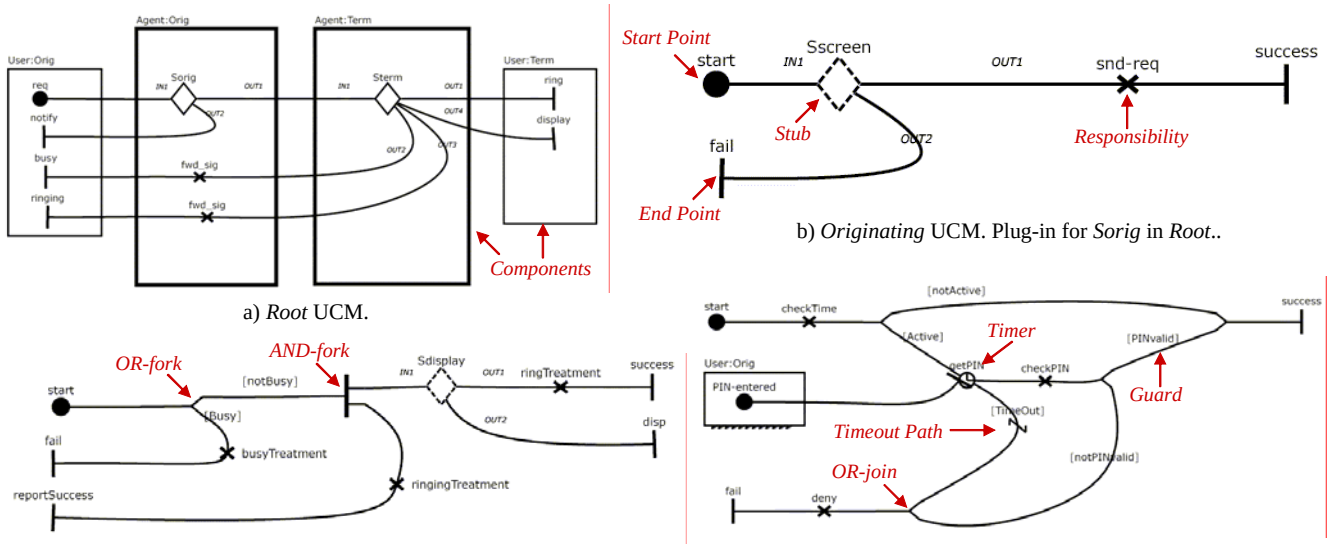
This section first presents a UCM example that covers most of the core notation elements. Then, several challenges related to scenario generation are discussed, together with an overview of our solutions.

2.1 Core UCM Notation Elements

The core elements of the UCM notation are recalled in a simple telephony example (Figure 2) explained in detail in [6]. In a nutshell, scenarios begin at *start points* representing preconditions and triggering events, and they culminate with *end points*, which describe postconditions and resulting events. *Responsibilities*, which can be allocated to particular *components*, are visited along the way. The scenarios are said to be causal because they involve partial orders of responsibilities and because they connect causes to effects. Guarded choices (*OR-forks*) represent alternative scenarios whereas *OR-joins* merge scenario paths. *AND-forks* introduce concurrent paths which can be synchronized using *AND-joins*. *Timers* set along a path can be triggered by the reaching of an end point connected to them, which cause the continuation of the scenario along the initial path. Otherwise, if the trigger is not provided in time, the *timeout path* is selected.

This example describes the connection phase of a simple agent-based telephony system. Originating and terminating users both have agents that handle their half of the call. Users can subscribe to various features such as Originating Call Screening, TeenLine (where a Personal Identification Number is required to make a call), Call Name Display, and so on. Figure 2a, which describes the top-level map, has stubs for each agent. A *stub* is a container for submaps, which are then called *plug-ins*. Plug-in UCMs are bound to their parent stubs by linking stub input/output segments to the start/end points in the UCMs. This ensures the continuity of scenarios across modular UCMs. A subset of the plug-ins is shown in Figure 2 (b, c, d), and the binding relationships are detailed in Figure 3. Stubs are *static* when they have a single plug-in (e.g. `Sorig`) and they are dynamic when they contain multiple plug-ins (e.g. `Sscreen`). Dynamic stubs are shown with dashed diamonds, and they contain a selection policy which enables the selection of one of their plug-ins at runtime, according to the system state.

UCMs contain a simple *path data model* that enables the selection of alternative paths and of plug-ins in dynamic stubs. A UCM model includes a set of user-defined Boolean variables that can be used to formally express conditions (pre- and post- conditions, guards, selection policies, etc.). *Scenario definitions* enable modelers to define scenarios in terms of initial values for these global variables, and in terms of the start points that are initially triggered (more than one start point can be triggered in parallel). When visited, responsibilities can also modify the content of the variables through *operations*. Figure 7 gives an example of scenario definition, and Section 3 will detail this important aspect of UCM traversal.



a) *Root* UCM.

b) *Originating* UCM. Plug-in for *Sorig* in *Root*..

c) *Terminating* UCM. Plug-in for *Sterm* in *Root*.

d) *TeenLine* UCM. Plug-in for *Sscreen* in *Originating*.

Figure 2 UCMs for a simple telephony example

Root UCM
  Sorig Stub
    Originating UCM. Condition: *True*. Binding:
    {<*IN1*, start>, <*OUT1*, success>, <*OUT2*, fail>}
      Sscreen Stub
        TeenLine UCM. Condition: *subTL*. Binding:
        {<*IN1*, start>, <*OUT1*, success>, <*OUT2*, fail>}
        OCS UCM. Condition: *subOCS*. Binding:
        {<*IN1*, start>, <*OUT1*, success>, <*OUT2*, fail>}
        Default UCM. Condition: ¬(*subOCS* ∨ *subTL*).
        Binding: {<*IN1*, start>, <*OUT1*, continue>}
  Sterm Stub
    Terminating UCM. Condition: *True*. Binding:
    {<*IN1*, start>, <*OUT1*, success>, <*OUT2*, fail>,
     <*OUT3*, reportSuccess>, <*OUT2*, disp>}
      Sdisplay Stub
        CND UCM. Condition: *subCND*. Binding:
        {<*IN1*, start>, <*OUT1*, success>, <*OUT2*, disp>}
        Default UCM. Condition: ¬*subCND*.
        Binding: {<IN1, start>, <OUT1, continue>}

Figure 3 Content of stubs, with conditions and bindings

The reader is invited to consult [1][3][10] for more detailed description of the notation as well as tutorial material. A recent survey on scenario notations also compares UCMs with fourteen other popular notations [2]. One of the conclusions is that notations such as UCMs and UML activity diagrams are useful and suitable for requirements engineering activities whereas MSCs and UML sequence diagrams are more adapted to detailed design activities.

2.2 Well-nestedness

In UCMs, OR-fork/join and AND-fork/join constructs can be used in any way to combine path segments while integrating scenarios. They can also fork or join more than two segments. Additionally, a UCM can have more than one start point and more than one end point. All of this increases the flexibility of the notation for the modeler, but this also makes the traversal of UCMs quite difficult.

One particular challenge is caused by UCMs that are not *well-nested*. A UCM can be seen as a graph where the nodes are the UCM constructs and the arcs the path segments connecting them. If this graph can be described in a linear way with operators for sequence (;), alternatives (+), and concurrency (|), without losing any of the causal relationships found in the original UCM, then this UCM is said to be well nested.

Figure 4a shows a well-nested UCM that can be represented linearly as K;L;((M;O)+(N;P)). The UCM in Figure 4b is also well-nested, as it can be represented as Q;R;((((S;(T|U))|X);W) + ((Y+Z);AA)). However, Figure 4c cannot be represented using such operators without losing some of the causal relationships.

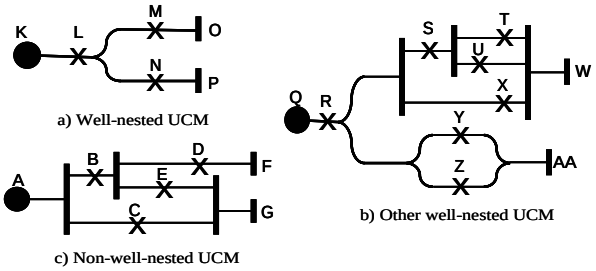


Figure 4 Well-nestedness and UCMs

Individual scenarios extracted from UCMs are partial orders and hence can include sequence (;) and concurrency (|) operators, but no alternatives (+). Accordingly, Figure 4a has two potential individual scenarios (K;L;M;O and K;L;M;P), while Figure 4b has three (Q;R;Y;AA, Q;R;Z;AA, and Q;R;((S;(T|U))|X);W).

For non-well-nested UCMs, different traversal strategies may lead to different resulting scenarios. For Figure 4c, a first strategy would lead to the scenario A;((B;E)|C);((D;F)|G), also represented as a well-nested UCM in Figure 5a. Such traversal may lose some of the original causal relationships and may also constrain the concurrency found in the source UCM. For instance, D can no longer be visited before C or E. Another strategy could lead to A;((B;((D;F)|E))|C);G, illustrated in Figure 5b. This time, although D can now be visited before C or E, this scenario adds new concurrency constraints (e.g., F can no longer happen after G). It is important to note that traversal mechanisms that extract individual scenarios from arbitrarily complex UCMs should not add concurrency or causal relationships not present in the source UCM.

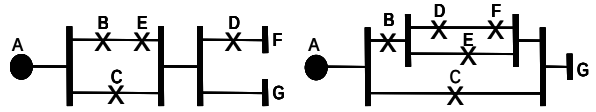


Figure 5 Two different traversal results for Figure 4c)

We must hence be aware that extracting individual scenarios as partial orders may lose some of the concurrency and causality information found in UCMs that are not well nested. One possible relief consists in applying different algorithms and then combining the resulting scenarios; tool architectures must be flexible enough to enable this.

In the draft UCM standard (Z.152) [10], no specific algorithm is enforced. However, a list of requirements for path traversal mechanisms is provided, and traversal algorithms must comply with these requirements. The requirements are flexible enough to support various depth-first, breadth-first, and hybrid traversals, as well as collapsing strategies. In Section 3, we give a brief overview of the new algorithm we prototyped in the UCMNAV tool.

2.3 Plug-ins and components

Another challenge we need to address is that of components inside plug-ins. UCM elements, and especially responsibilities, can be allocated to components, which may also contain sub-components. This is visually shown as UCM elements on top of components. For instance, the fwd_sig responsibility in Figure 2a is allocated to the component Agent:Orig (that is, an agent instance in the originating role). UCM path elements not allocated to components are part of the environment.

Plug-in UCMs can also contain components. The semantics is that the components and path elements in a plug-in UCM are allocated to the component containing the parent stub. For example, snd-req in Figure 2b, which is a plug-in bound to stub Sorig in Figure 2a, is therefore allocated to component Agent:Orig. This is applicable at any level of nesting: checktime in Figure 2d also belongs to component Agent:Orig (TeenLine belongs to Sscreen, and Originating to Sorig).

Anchored components, which are shown with small diagonal lines (shadow) under the component, are handled differently. These are references to components outside the current scope. For instance, User:Orig in Figure 2d makes reference to a component at the same level as Agent:Orig (see Figure 2a), i.e. User:Orig is not contained inside Agent:Orig.

When extracting scenarios, the traversal mechanism should associate the visited elements with the right component.

```
<!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)*>

<!ELEMENT do EMPTY>
<!-- WP_Enter: When PT gets to a waiting place
WP_Leave: After the waiting place is visited
Connect_Start: Start point of a plug-in
Connect_End: End point of a plug-in
Trigger_End: Connected End point.      -->
<!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 describing
the condition. -->
<!ATTLIST condition
   hyperedge-id    NMTOKEN     #REQUIRED
   label           CDATA       #REQUIRED
   expression      CDATA       #IMPLIED >
```

Figure 6 DTD for Scenarios Extracted from UCMs

2.4 Scenario representation

Another challenge is the choice of the representation format for the resulting scenarios. In [14], the authors selected an existing notation (MSCs) because of the popularity of this standard notation and because it was compatible

with tools (e.g. Telelogic TAU). However, it becomes very difficult to convert such MSC scenarios to other presentation formats because it contains many details not relevant to other notations (e.g. messages, which need to be created artificially along the way), and also it does not capture all the semantics of UCM scenarios.

The need for a standalone scenario representation that captures UCM semantics and that is convenient for further transformations to other scenario languages led us to define an XML-based format, whose Document Type Definition (DTD) is described in Figure 6. This DTD essentially formalizes the syntax of the scenario grammar used in Section 2.2, i.e., a scenario supports the recursive use of sequence (`seq`) and parallel (`par`) operators. Scenarios can also be parts of a `group`, and XML files compliant with this DTD can also include many groups. This reflects the structure of scenario definitions found in UCMNAV (see Figure 7). The `condition` element captures the conditions satisfied during the traversal of the UCMs (e.g. at choice points and in dynamic stubs). The `do` element, which can be of various types, describes each UCM element visited together with the component to which it is allocated (if any, as suggested in Section 2.3).

All these elements have attributes that preserve traceability to the UCM elements (called *hyperedges* in the UCM internal representation) and components found in the original UCM file, which is also stored in XML. Other attributes (`name`, `description`, etc.) replicate some of the information found in the original UCM file in order to minimize the need to access the latter during transformations to target scenario languages.

3. Scenario definitions

UCM scenario traversal requires three components: UCMs, scenario definitions, and a traversal algorithm compliant to the requirements described in [10]. The output is a collection of scenarios where sequences and concurrency are preserved (at least partially in the case of non-well-nested UCMs), but where alternatives are resolved. These scenarios are stored as XML files valid with respect to the DTD shown in Figure 6. All this is part of the first step of our scenario generation approach (see Figure 1). UCMNAV, already used for creating and navigating UCMs and for storing them as XML file, was augmented to support our new traversal algorithm which produces scenario files in XML.

Figure 7 (top) shows the interface used to create scenario definitions, which are organized in groups. For each scenario, one needs to provide a name, the list of start points to be triggered, the initial values of the global Boolean variables, and (optionally) a post-condition used to assert the validity of a scenario once the traversal has completed. This interface shows buttons used to produce XML files for individual scenarios or for whole groups. We can also generate MSCs directly, but this is done with the older, problematic transformation algorithm from [14], which was left available in the tool for comparison purpose. The traversal algorithm is also used for highlighting the visited path in a different color and to restrict the navigation of the UCMs to that selected scenario (bottom of Figure 7). This functionality, which is useful for the understanding, exploration, and analysis of complex UCMs, was available in [14] but now it uses our new algorithm.

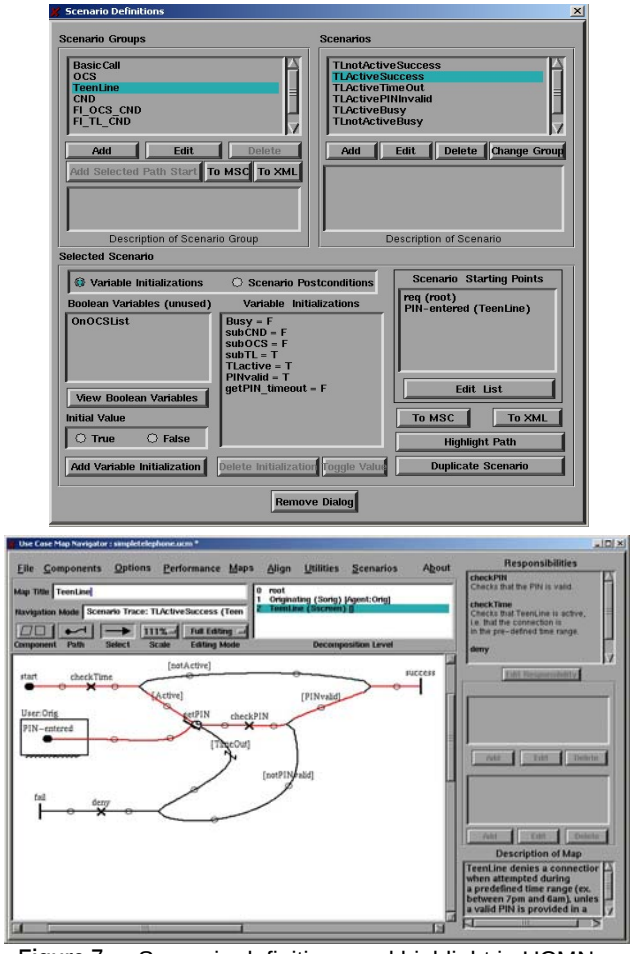


Figure 7 Scenario definitions and highlight in UCMNAV

The new traversal mechanism we prototyped has two phases: the first constructs an intermediate traversal graph while traversing the UCM according to the scenario definition, and the second collapses this graph to make it well-nested. Due to the complexity of these two phases, we can only provide an intuitive overview in this paper.

This first phase uses three data structures that accumulate some of the elements that are partially visited, i.e. whose *Path Continuation Condition* (see [10]) is not yet entirely fulfilled: *Fork_Stack* contains the AND-forks whose outgoing branches have not all been visited, *Path_Elements* stores elements whose incoming branches have not all been visited (e.g. AND-joins, timers, and waiting places), and *Start_Points* initially contains the list

of start points found in the scenario definition. *Fork_Stack* and *Start_Points* may be implemented as LIFO queues or FIFO stacks: this choice will depend on the traversal strategy (depth-first, breadth-first, or some hybrid solution).

The new algorithm we implemented in UCMNAV uses a depth-first traversal of the graph that captures the UCMs' structure. It is depth-first because it keeps traversing the path elements until a stop point (AND-join, waiting place, or timer) is reached, then it backtracks to get the next available branch of an AND-fork (in *Fork_Stack*) or the next start points (from *Start_Points*). This approach treats these three types of stop points in a very similar way, which is simpler and more robust than older algorithm used in [14]. The traversal is successful if *Fork_Stack* and *Start_Points* are empty at the end, the items in *Path_Elements* are all marked as visited, and the scenario definitions' post-condition expressed as a logical predicate on the Boolean variables evaluates to true. Otherwise, the algorithm fails with an appropriate error message. Plug-ins are traversed via their connections to stubs; a stub whose plug-in is not correctly bound will stop the progression of the traversal and result in an error. The algorithm can also detect infinite loops (through a maximum number of visits). It can also iterate on all scenarios of a group and on all groups if required.

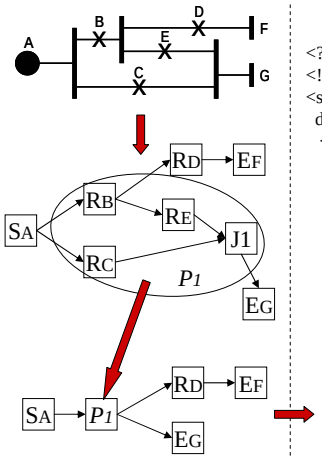
The second phase takes the result of the first phase (an intermediate graph structure) and manipulates it to make it well-nested. Figure 8 reuses the UCM from Figure 4c as an example. For a scenario definition starting with A, the first phase produces the graph found in the middle. R nodes are responsibilities, S/E nodes are start/end points, and J nodes are synchronization (AND-joins, timers, waiting places). The collapsing algorithm goes as far as it can according to the strategy used in phase one. When it has visited all the input segments of a J node, it backtracks to the original forking node (e.g. SA) and then collapses everything in between, which results in node P1. The segments leaving the ellipse in the middle figure remain connected to P1 in the resulting graph (bottom figure). This is repeated recursively until there is nothing left to collapse. The collapsed graph is then converted to XML: single arrows are converted to sequences (`seq`), and collapsed nodes and multiple arrows to parallel statements (`par`).

4. Scenario transformations

The second step of our generation technique consists in transforming the XML scenario file into the target scenario language (see Figure 1). The tool used for this transformation is called the CONVERTER and is separate from UCMNAV. The CONVERTER is composed of three main components. An XML parser (#6 in Figure 1) first validates the source scenario file against the scenario DTD. We use the XERCES parser from the Apache project [16]. The Scenario Extractor (#7) then extracts individual scenarios from the validated scenario file, which may contain many groups and scenarios per group. An XLST processor (#9, XALAN [16]) is finally used, in combination with XSL transformation rules (#8), to transform each individual scenario into a target scenario file (#10). The CONVERTER can also consider custom rules (#11) that override the default transformation rules provided (#8).

An interesting property of this architecture is that a library of transformations can be implemented as a collection of XSL files, independently from any work on the UCM and UCMNAV side (see the shaded box in Figure 1). The scenario XML files (#3) hence shields the traversal mechanism from the transformation mechanism.

Figure 9 presents an extract of the default XSL transformation rules (#8) for generating MSCs from XML scenarios. This XSL file contains about 1000 commented lines that provide templates for each of the elements in the scenario DTD and for each of the sub-types of the `do` element. Each of these templates can be overridden by custom rules (custom.xsl). Several additional functions for handling identifiers and counters are also provided.




```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE scenarios SYSTEM "scenarios1.dtd">
<scenarios date ="Sun Mar 23 21:44:46 2003" ucm-file = ".ucm"
 design-name = "" ucm-design-version = "2">
 <group name = "MyGroup"  group-id = "1" >
  <scenario name = "QSICfigure4c" scenario-definition-id = "1" >
   <seq>
    <do hyperedge-id="0" name="A" type="Start" />
    <par>
     <seq>
      <do hyperedge-id="16" name="B" type="Resp" />
      <do hyperedge-id="25" name="E" type="Resp" />
     </seq>
     <seq>
      <do hyperedge-id="19" name="C" type="Resp" />
     </seq>
    </par>
    <par>
     <seq>
      <do hyperedge-id="22" name="D" type="Resp" />
      <do hyperedge-id="15" name="F" type="End_Point" />
     </seq>
     <seq>
      <do hyperedge-id="2" name="G" type="End_Point" />
     </seq>
    </par>
   </seq>
  </scenario>
 </group>
</scenarios>
```

Figure 8 Collapsing of an intermediate graph into a well-nested graph, and XML representation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/
  1999/XSL/Transform" version="1.0">
<xsl:import href = "custom.xsl"/>
<xsl:output method = "text" />
<xsl:strip-space elements = "*" />

<!-- Template for 'Start' attribute; generates first message. -->
<xsl:template name = "Start">
<xsl:param name = "msgid" />
   <!-- print all 'following-sibling' cond. -->
   <xsl:call-template name="write_conditions" />
   <!-- print out messages and proc next following node -->
   <xsl:call-template name = "generate_message">
      <xsl:with-param name = "msgid" select = "$msgid" />
   </xsl:call-template>
</xsl:template>

<!-- Matched template for the root 'scenario' element. -->
<xsl:template match = "//scenario">
   <xsl:call-template name = "write_document_header" />
   <xsl:call-template name = "generate_instances" />
   <xsl:call-template name = "write_scenario_header" />
   <!-- apply template for first child node -->
   <xsl:apply-templates select = "child::*[1]" >
      <xsl:with-param name = "msgid" select= '1'/>
   </xsl:apply-templates>
   <xsl:text>&#xa;</xsl:text>
   <xsl:call-template name = "generate_endinstances" />
   <xsl:call-template name = "write_document_footer" />
</xsl:template>
```

Figure 9 Extract of XSL transformation rules for MSC generation from XML scenarios

In this particular transformation, the templates handle our default mapping of UCM individual scenarios to MSCs. Start points and end points map to messages coming from or going to the environment, responsibilities become actions, conditions become MSC conditions, timers become MSC timers, and connect_start and connect_end (in plug-ins) are ignored because they are simply used as connectors. The parallel element is also preserved through the MSC *par* inline statement. UCM components become MSC instances.

One of the main challenges is the handling of inter-component causality, which is refined in MSC using messages. Since UCMs do not contain any information relative to the message exchanges required to implement causal relationships across components, synthetic messages (*m0, m1, m2, ...*) have to be inserted. These abstract messages could be refined into more detailed and realistic protocol exchanges given the necessary information, and this could be done in custom XSL rules.

An example MSC generated by the CONVERTER is shown in Figure 10. This MSC results from the definition of a basic call successful scenario for the simple telephony UCM in Figure 2. One can notice the presence of messages *m1* and *m2* to ensure that the causal relationships between the originating and terminating agents are preserved. Our CONVERTER generates MSCs in textual form (according to the Z.120 Recommendation [8]), and the graphical representation is produced with a MSC viewer (e.g. Telelogic TAU). Such graphical view greatly enhances the understandability of long scenarios, which otherwise would span multiple UCMs (e.g. many plug-ins).

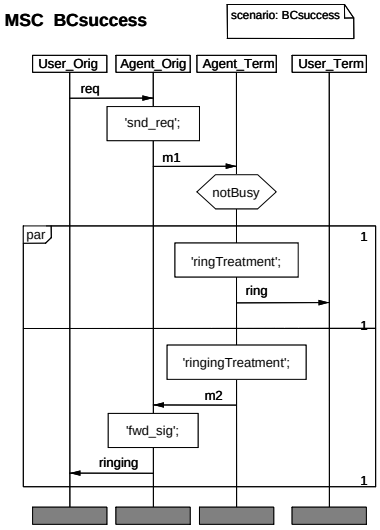


Figure 10 MSC generated from the CONVERTER

5. Discussion

The idea of separating the source representation from the target presentation has great significance. First, tool development becomes easier as UCMNAV and the CONVERTER can evolve independently, as long as they agree on the DTD for the XML scenarios. Second, the development of different transformations for different target scenario representations will promote the usage of UCMNAV as a requirements engineering and design tool, which will in turn result in better support. Third, we hope the availability of various target scenario formats will encourage research on UCMs. Already, we are working on new transformations to UML sequence diagrams (in XMI) and to validation test cases in the ITU-T TTCN-3 language.

Generating scenarios from UCMs promotes validation of requirements and designs at three levels: the traversal process can stop due to incorrect use of conditions and paths in the UCM, the resulting scenarios can be used to identify unexpected emergent behavior and conflicts in UCMs, and these scenarios can further be used as test goals for the detailed design and the implementation.

A recent application of this process which involves tool integration is presented in [6]. The MSCs generated from the UCMs are used to automatically synthesize an SDL model [7][12], which is executable and amenable to formal validation and detection of problems such as unspecified receptions, deadlocks, livelocks, etc. Many restrictions to the synthesizer used in that experiment [11]

can be overcome by providing custom transformation rules in XSL (#11 in Figure 1).

Our approach shares some similarity with that of Gu and Petriu [4], who explored the XSLT-based transformations of UML activity/sequence diagrams to Petri Nets. Our generation process is less ambitious as it focuses on simpler scenarios, but it is more flexible because many target languages can be easily supported. In her thesis, Guan also explored the automated generation of models (in LOTOS) from UCMs [5]. These scenarios must be expressed using a subset of the UCM notation (e.g. no timers). Her algorithm translates the whole UCM model at once, without using scenario definitions or conditions that involve the global variables. As a result, many invalid scenarios can emerge in the target LOTOS model.

6. Conclusions

The UCM notation's capability of integrating many scenarios in a single view is one of its strengths and is suitable for understanding, analyzing, and reasoning about the system as a whole. However, individual scenarios are still an important aspect, especially as we move towards the detailed design phase and the testing phase. This paper presented a new approach to the generation of individual scenarios from UCMs. It decouples the traversal based on scenario definitions from the transformation to target scenario languages. Intermediate scenario files in XML format are used as an interface between scenario representations and scenario transformations. This work was implemented partly in the UCMNAV tool, and partly in a new CONVERTER tool that combines off-the-shelf XML parser and XSLT engine with a small scenario extractor. Section 5 recalls many benefits of our approach over the current state of the art. Throughout the article, many examples were provided to illustrate the main challenges faced in such a transformation as well as our solutions.

Although good progress was achieved, many improvements are still possible: Better handling of multiple start points in scenario definitions (e.g. multiple triggering of one start point); definitions of reusable sub-scenarios; better handling of plug-in instances; support for user-selectable traversal algorithms in UCMNav; and XSL files for other target languages (e.g. UML sequence diagrams in XMI, and TTCN-3 test cases). We are currently investigating yet another intermediate step where messages would be created in an abstract way and stored in XML, before the final generation of MSC, UML, or TTCN (which would then be simplified). We expect this future work to also impact the development of the UCM notation itself (one of us, Amyot, is ITU-T Rapporteur/Editor for the URN standard). Case studies where this technique will be applied are also planned.

The authors thank G. Mussbacher for his contributions to the traversal guidelines, A. Miga for creating UCMNAV, S. Cui for fixing several bugs with the UCM-NAV-based generation process, A. Williams for his collaboration on the synthesis of SDL models, and the URN Focus Group for their efforts in making UCMs a useful standard. This work was supported financially by Nortel Networks, NSERC, and the University of Ottawa, Canada.

7. References

[1] D. Amyot: Introduction to the User Requirements Notation: Learning by Example. *Communication Networks*, 42(3), 285-301, 2003.

[2] D. Amyot and A. Eberlein: An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunication Systems Journal*, 24:1, 61-94, 2003.

[3] R.J.A. Buhr: Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*. Vol. 24, No. 12, Dec. 1998, 1131-1155.

[4] G. Gu and D. C. Petriu : XSLT Transformation from UML Models to LQN Performance Models. *WOSP'2002*, pp.227-234, Rome, Italy, July 2002.

[5] R. Guan: From Requirements to Scenarios through Specifications: A Translation Procedure from Use Case Maps to LOTOS, M.Sc. thesis, Univ. of Ottawa, Canada, Sept. 2002.

[6] Y. He, D. Amyot, and A. Williams: Synthesizing SDL from Use Case Maps: An Experiment. *11th SDL Forum*, Stuttgart, Germany, July 2003. LNCS 2708, 117-136.

[7] ITU-T: Recommendation Z.100, Specification and Description Language (SDL). Geneva, Switzerland, 2000.

[8] ITU-T: Recommendation Z.120 (11/99) Message Sequence Chart (MSC). Geneva, Switzerland, 2001.

[9] ITU-T: Recommendation Z.150, User Requirements Notation (URN) – Language Requirements and Framework. Geneva, Switzerland, 2003.

[10] ITU-T, URN Focus Group: *Draft Rec. Z.152 – UCM: Use Case Map Notation (UCM)*. Geneva, Switzerland, Feb. 2002. http://www.UseCaseMaps.org/urn/

[11] KLOCwork Corporation: *KLOCwork MSC to SDL Synthesizer Tutorial*, Version 1.0, 2002.

[12] N. Mansurov and D. Zhukov: Automatic synthesis of SDL models in use case methodology. *Ninth SDL Forum (SDL'99)*, Montréal, Canada, 1999.

[13] A. Miga, *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Canada, 1998. http://www.UseCaseMaps.org/tools/ucmnav/

[14] A. Miga, D. Amyot, F. Bordeleau, C. Cameron, and M. Woodside: Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. *Tenth SDL Forum (SDL'01)*, Copenhagen, 2001. LNCS 2078, 268-287

[15] OMG, Unified Modeling Language Specification (UML), version 1.5, March 2003.

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

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

[18] W3 Consortium, *XSL Transformations (XSLT) Version 1.0*, W3C Rec., 16 November 1999.