UCMExporter: Enabling UCM-Based Transformations

Report for the summer 2003 work term

By Ali Echihabi
Student Number: 2090356
Program: Software Engineering

Table of Contents

Table of Figures

Glossary

GUI GUI stands for Graphical User Interface. Most programs are now providing a graphical interface with windows and buttons to improve their usability.

MSC The Message Sequence Chart is the International Telecommunication Union's (ITU) standard for instance interactions in a system. It is very similar to SD (below).

SD The Sequence Diagram is a scenario notation that is part of UML (below). It describes the dynamic behavior of a system by graphically showing the interactions between the system components.

SDL The Specification and Description Language represents in a graphical format real-time and event-driven systems. It is widely used in the Telecommunication industry.

TTCN The Testing and Test Control Notation is used to write detailed test descriptions. TTCN is used to test a wide variety of systems such as wireless networks and mobile phones [8].

UCM Use Case Map is a requirements-gathering scenario-based notation that is mostly used for distributed systems. It is most suitable for the requirements gathering phase than other scenario notations.

UML The Unified Modeling Language is the industry standard for designing and documenting software components.

XMI The XML Metadata Interchange format (XMI) is a format that enables easy interchange of metadata (information about the data) between modeling tools (based on UML) and between tools and metadata in distributed heterogeneous environments [6].

XML The eXtensible Markup Language (XML) is a flexible and very simple text format that enables the exchange of a wide variety of data on the web or elsewhere [7].

XSL/XSLT XSL or XSLT is the eXtensible Stylesheet Language which defines how an XML document is to be transformed or displayed.

Acknowledgement

I would like to take this opportunity to thank Dr. Daniel Amyot for having selected me as his co-op student for the summer of 2003 and making me part of his group.
I would like as well to thank Jacques Sincennes and Yong He for their technical and theoretical help and advice.
Finally, I thank the co-op office of the University of Ottawa for their commitment and hard-work.

1. Abstract

In this report, I summarize my work with Dr. Amyot during my summer 2003 co-op work term at the University of Ottawa.
My project consisted of implementing a tool to transform Use Case Map (UCM) scenarios to Sequence Diagrams in the XMI standard. However, the project's scope grew to include previous work and to plan for future similar transformations. I gave my project the name of UCMExporter.
The UCM notation is used at the early requirements gathering phase of the Software Engineering Process in order to validate the functional requirements. It is most useful for high-level modeling of real-time distributed systems. Being able to transform UCM scenarios to other scenario notations such as MSC and Sequence Diagrams facilitates the transition from the requirements phase to the design phase. The goal of UCMExporter is to enable UCM-based transformations and group them under one tool.
In this report, I address the problems and challenges faced in my project. I also illustrate the important concepts and give a simple case study to show how to use UCMExporter.

2. Introduction

2.1 Scenarios and Use Case Maps

Use Case Map (UCM) is a modeling notation that helps software engineers think of and express their system at a high level of abstraction [1]. UCM models define the responsibilities at each step of a possible execution path of the system. An execution path, which we refer to as a *scenario*, defines both the responsibilities for a normal (expected) execution, and those for an execution with exceptions. The scenarios presented graphically are enough for the client to confirm the requirements of the system without going too much into the details (e.g. no component definitions) [1].
There are other scenario notations that are used in the industry. The most commonly known are the Message Sequence Charts (MSC) and Sequence Diagrams (SD). While both these notations describe a possible scenario, the software engineer needs to decide about several lower-level issues (e.g. components involved, communication patterns). Such decisions are necessary for building the system, however not at the initial step of gathering the requirements from the user. This is why it was found that the UCM notation was more adequate for the analysis phase and the other notations more suitable for the design phase [2].
The UCM notation allows us also to bind the scenario responsibilities to components in the system. This is especially important because the notation is mostly used for distributed systems [2]. A component in the UCM notation is very abstract. It can be anything from a person to a computer hard-drive. Moreover, the UCM notation is used at such a high level of abstraction that the software engineer does not need to worry about the communication between the components.
The strength of the UCM notation for the modeling of distributed systems is gradually being accepted and their usefulness appreciated by the academic and industrial sectors. There is now a UCM Users' Group [5] website that promotes the use of UCM and publicizes the related news and events. Dr. Amyot is currently the chair of the UCM Users' Group.
UCM models can be defined using the open-source tool UCMNav (UCM Navigator). By using this tool the software engineer can capture the requirements from the client, and define the possible scenarios of execution in the system. The following two figures show a sample Use Case Map and a possible scenario defined with UCMNav:

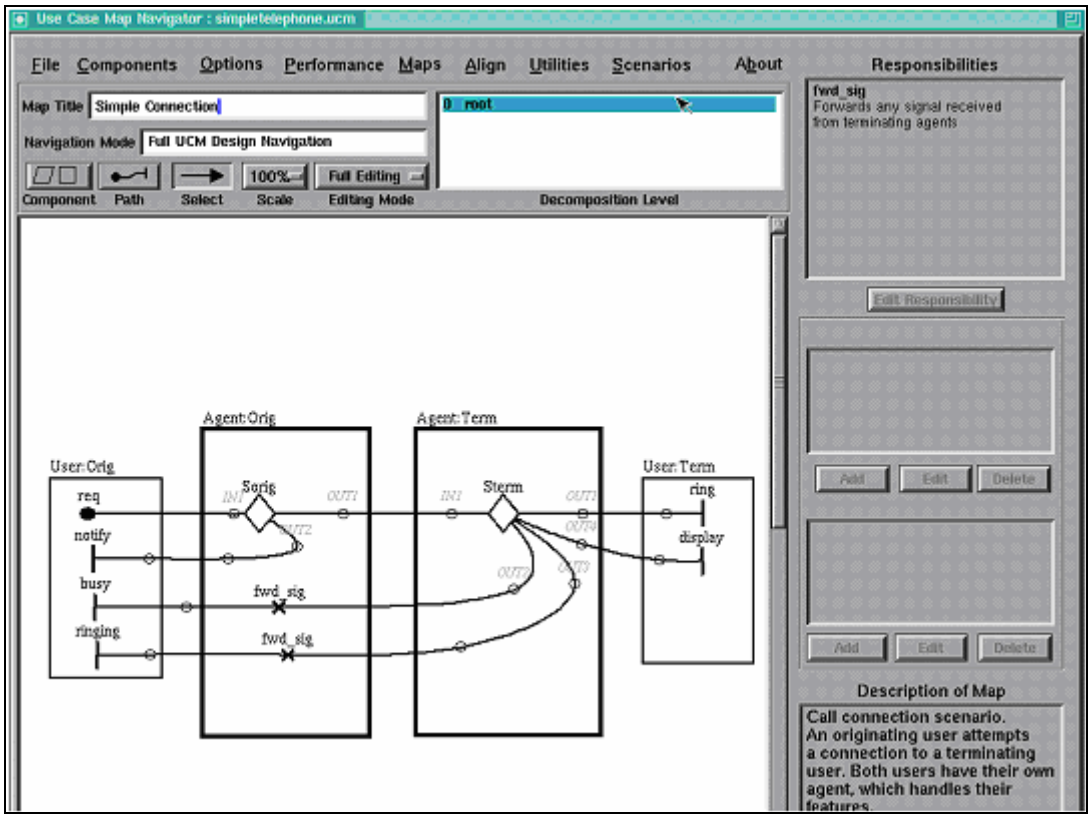

Figure 1 - Sample UCM defined using UCMNav

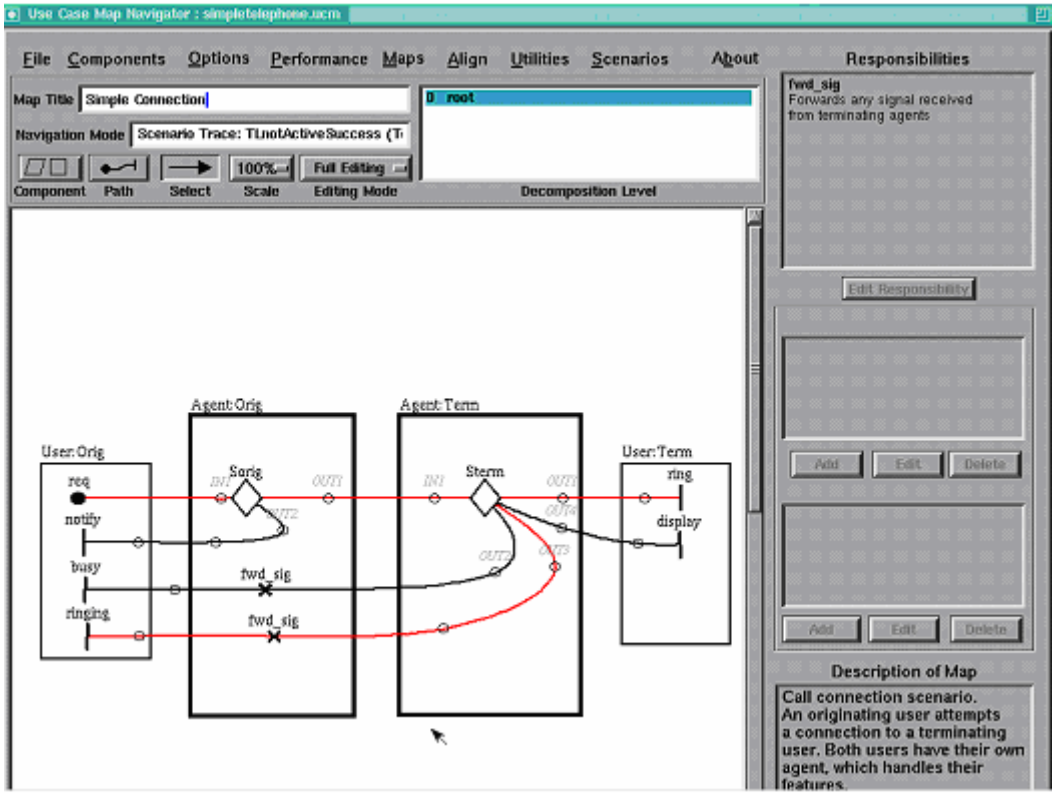

Figure 2 - Sample Scenario shown as a highlighted path (in red)

As stated above, UCM models are more suitable for the requirements phase, but other scenario notations (e.g. MSC) along with their tools are more suitable and more widely used for the design phase. Both phases are necessary for the development of a well-structured and robust system. Hence, it is important to have tool interoperability in order to transfer the time and effort put in one phase to the next.

2.2 Research Motivation and Past Work

Providing the tool support for the UCM notation and enabling the UCM-based transformations to other scenario notations was a major goal for Dr. Amyot and the members the UCM User's Group.
A lot of effort was put to implement, maintain, and evolve UCMNav. One feature that was added to the original UCMNav was to generate an MSC by doing a UCM scenario traversal. This feature was very limited and inflexible to generate another scenario notation from the UCM scenario traversal [2]. It was also hard to maintain and error-prone because it was traversing the scenario and generating the MSC at the same time.
In order to solve the problem of the programming complexity and inflexibility, a two step approach was devised by Dr. Amyot and collaborators [2]. The first step consisted of generating a description of the scenario traversal in XML format. The second step consisted of taking the XML scenario traversal description and transforming it into another scenario notation (MSC for instance).
The separation of the scenario traversal and the transformation steps allowed for easier transformation algorithms that were based on the XML description. The use of XML as a description format meant that all transformation programs knew the format and structure of their input, and could then be programmed separately by different programmers using different programming languages.
Two of Dr. Amyot's past students wrote XSL programs to transfer the XML input into TTCN and MSC files. They used XSL to implement a predefined mapping between UCM concepts and test case concepts for TTCN and message and action concepts for MSC. The TTCN and MSC files were generated in a format that was supported by the most widely used tools in industry. Their work showed that the UCM scenario traversal description in XML format contained enough information in order to generate a description of the system's UCM model in another modeling language.
By allowing the transformations from UCM to TTCN and MSC, a step was made towards the interoperability of tools and the transfer of effort from the analysis phase to the design phase.

3. Project

3.1 A Look at the Previous Work

3.1.1 UCM2TTCN

UCM2TTCN was the name I gave to the application developed by Bryan Mulhivill which generated test cases and test goal descriptions from the XML description of a scenario. Bryan worked under the supervision of Dr. Amyot and wrote a C++ program that applied an XSL sheet on the XML input to get the sought after result. The figure below shows the black-box view of UCM2TTCN:

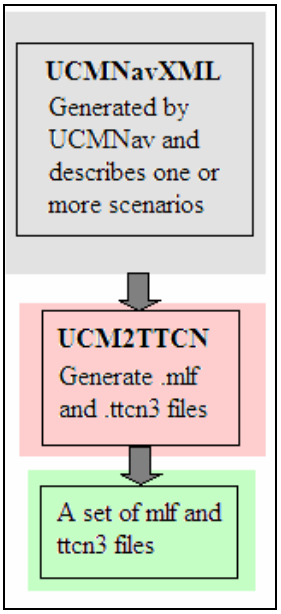

Figure 3 – Black-box view of UCM2TTCN

3.1.2 UCM2MSC

UCM2MSC was the name I gave to the XSL sheet written by S. Cui to transform the input XML into an MSC file. The generated MSC files were in a textual format that could be visualized with Telelogic Tau 4.4. Below are two figures showing the black-box view of UCM2MSC and a sample MSC:

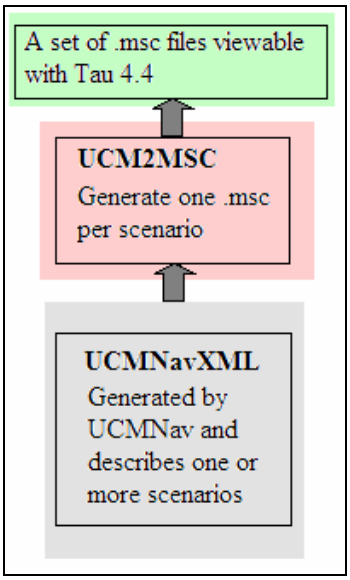

Figure 4 – Black-box view of UCM2MSC

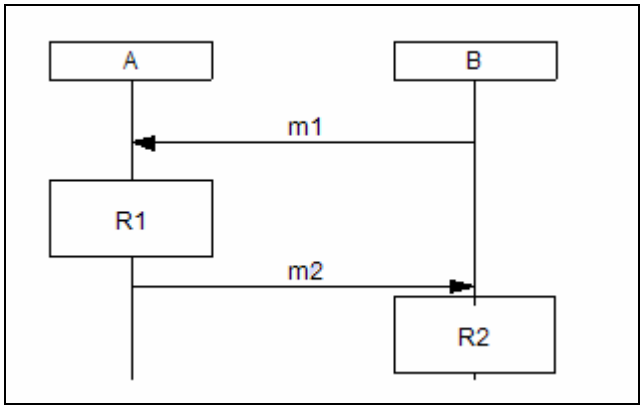

Figure 5 - Sample MSC

Cui defined a mapping between UCM concepts (such as responsibilities) and MSC concepts (such as actions and messages). Cui implemented the mapping using XSL only which made the XSL sheet fairly complicated. I assessed the complexity of the XSL sheet when I had to fix a few defects in it. It became clear to me that XSL was not appropriate for complicated and logic intensive programming (as Cui had to do) but rather suitable for quick and predefined transformations.

3.2 UCM2XMI : My project's initial scope

3.2.1 Description

UCM2XMI was the name I gave to my initial project. It consisted in the beginning of defining an XSL sheet like Cui's in order to generate a Sequence Diagram (SD) in the XMI format for each input scenario description (in XML format). Sequence Diagrams are UML's popular notation to describe the dynamic behavior of a system. Hence it was important to provide the transformation between UCM scenarios and SD. XMI (XML Meta-model Interchange) is a standard format of UML models. Theoretically, this standard would allow complete interoperability between UML tools. This meant that the result of UCM2XMI would be compatible with all tools that respect the standard. We refer to the Sequence Diagrams in XMI format as XMI/SD. Below is a figure showing a sample XMI/SD:

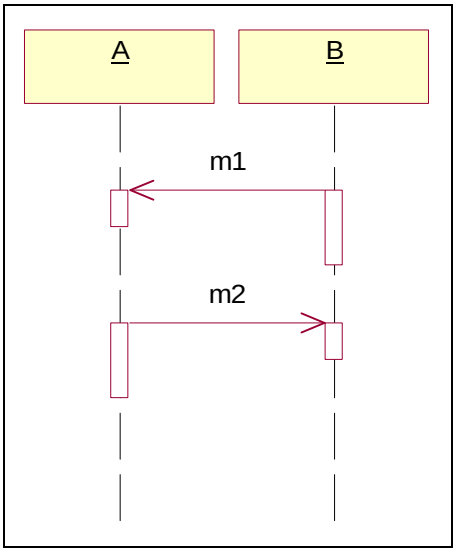

Figure 6 - Sample UML Sequence Diagram

3.2.2 Problems and Challenges

Initially, UCM2XMI seemed to be a close replication of the effort done for UCM2MSC since their outputs (Figure 6 and Figure 5 respectively) had many things in common (such as instances). The similarity between the two helped save time in the initial mapping between UCM and SD concepts by reusing the mapping between UCM and MSC. Unfortunately, UCM2XMI faced problems and challenges that made it much harder to implement and test. I explain the important challenges and problems faced below:

a) The Layout of the Sequence Diagram:

The XMI (version 1.1) standard did not define how to represent the layout information, but left this matter as tool specific. Moreover, no tool on the market at the time we started the project could do an auto-layout starting from a correct XMI file. This meant that we could not visualize the XMI output as defined by the standard, and that was a major problem because all testing and validation could only be done visually (being able to answer the question: does it look correct?).
Something that made this problem even harder was the lack of documentation. Even though we approached some tool vendors concerning this issue, we did not find the documentation that described how a tool specifically did its diagram drawing. Therefore I had to find myself how tools drew their diagrams.
Since that was clearly a time consuming effort, we decided to limit our support to the most widely used tool in industry and that was Rational Rose. For a period of almost a month, I analyzed the diagrams generated by Rational Rose and modified them in order to obtain the format that the output of UCM2XMI needed to obey.
After that period, I was able to manually write XMI files and add diagram information to them and successfully visualize them with Rational Rose. While that opened the door for the implementation, it stayed a major concern for us because my output would be tool specific and would depend on non-documented rules that I learned from examples. This meant that my output could not be visualized with other tools, and that it could be affected by unadvertised changes in the internal workings of Rational Rose.

b) Expressing the Parallelism:

Concurrency, or *parallelism*, is a common and powerful behavior of real-time and distributed systems that the UCM notation is mostly used for. Parallelism means that two responsibilities can be executed in parallel (at the same time) in either one or separate components. Parallelism is not limited to two sequences. It can happen among several sets of responsibilities, each set having responsibilities happening in sequence or in parallel themselves. The idea behind parallelism is not to impose an order (sequence) between sets of responsibilities, but rather let them execute at the same time each at its own speed. Generally, we cannot expect the order of the execution of the parallel sequences. Hence, drawing an SD or MSC (as shown in Figure 6 and Figure 5 respectively) does not express the parallel aspect of execution but rather captures only one of the many possibilities.
The MSC notation has been augmented to visually express the concept of parallelism. A sample parallel scenario is expressed in MSC in Figure 7. However, SD lacks this concept (It is expected to be added in the next version of UML).

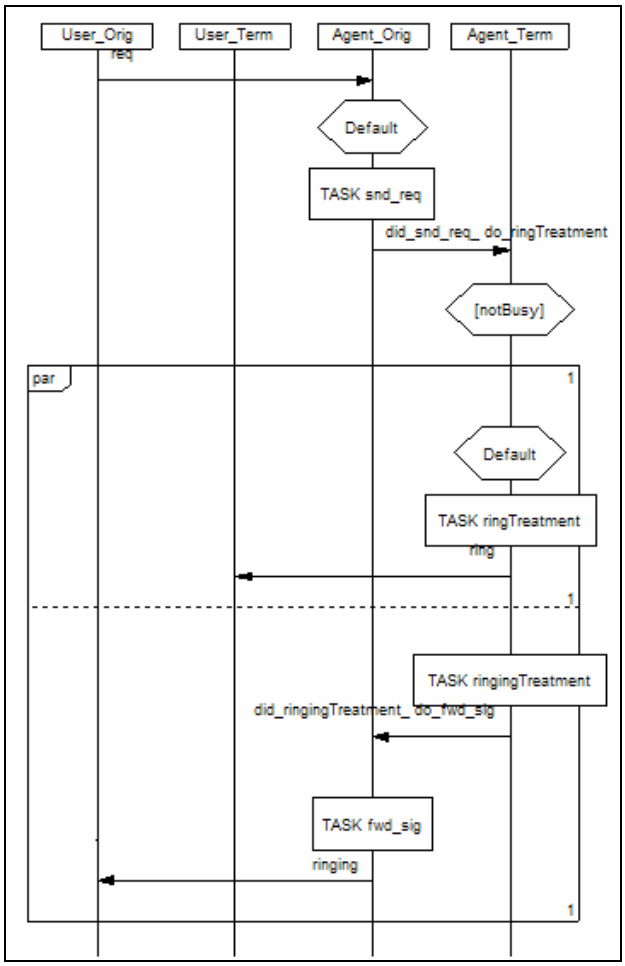

Figure 7 - A sample parallel scenario in MSC notation

In order not to loose the (important) parallel aspect of a UCM scenario when we translate it to XMI/SD, I tried three methods to show that aspect visually. The three methods were: putting comments in the SD to mark the beginning and end of a parallel block, using colors to show what is parallel with what, or using labels (text) to do so.

In the first method, I tried to express parallelism using comment blocks that could be put as markers of the beginning and end of a parallel block. For one parallel block, this method would yield a result similar to removing the “par” box from the MSC in Figure 7 and putting small comment boxes at the top left and bottom left corners. Unfortunately, I could not implement this method because the XMI importer Rational Rose used (and that was Unisys) discarded the location of the comment boxes and put them in a default location. Even if this method had worked, it would still be limited in expressing multi-level nested parallelism (it would overcrowd the SD with comment boxes).

The second method aimed at defining a coloring scheme that would express which messages happened in parallel with which and which happened in sequence. Before going too far with this method, I tested the Unisys XMI importer that Rational Rose used and I found that it discarded the color information too and would use its default color instead. This method would be limited as well even if it had worked, because there was no intuitive way to express nested parallelism.
Being left with the third method only, I tried to formulate a labeling mechanism for messages in order to express their parallelism relationship. The idea being that from a simple label added to the message name, the software engineer would be able to determine what was in sequence with that message and what could be in parallel, and also at what level of nestedness that message took place.
The labeling method worked very well to answer our needs. In Figure 8, I am giving the equivalent SD of the MSC that we looked at in Figure 7.

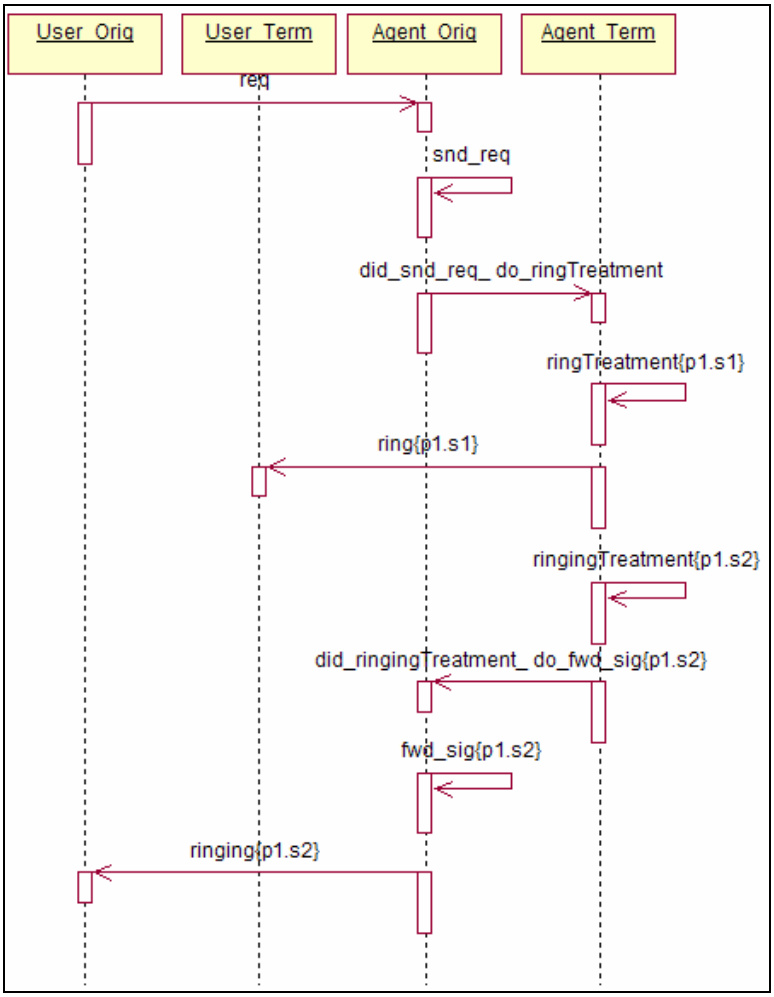

Figure 8 - Parallelism expressed in UML/SD

In the above figure, the label “p1.s1” in the messages means that these messages belong to the first sequence of the first parallel block, and hence happen in the specified order. The same interpretation holds for “p1.s2”. Messages labeled “p1.s1” are in parallel with messages labeled “p1.s2”. This method was successfully tested with parallel blocks having more than two sequences with nested parallelism. To give an example, the label for a message in the first sequence of the first parallel block nested inside the second sequence of the first parallel block in the scenario would be “p1.s2.p1.s1”.

The labeling system used was very easy to get used to and expressed the parallelism relationship of any message with respect to the rest of the scenario.

3.2.3 Architectural Choices

In order to cope with the problems and challenges explained in the previous section, I had to make an architecture that would allow for easy maintenance and evolution.
Because the output would be tool dependent and affected by future changes in the standard or the internal workings of the tool, I broke the process into two steps. In the first step, I transformed the input XML scenario description into XMI that strictly corresponded to the standard (version 1.1). The second step was responsible for taking the generated XMI of the first step and adding the diagram information to it.
The reason of having a step that would generate standard XMI was that we have hoped that there would be a tool that could do the diagram auto-layout. The second step separated the generation of XMI and the final tool dependent output. Changes in the internal workings of the tool would necessitate modifying this second step alone. While this step supported Rational Rose only, it was designed to independently support other UML tools as well.
Another important architectural choice I made was the separation and coherence of the logic, i.e. I have separated things that were related and grouped things that were. Because the transformations were logic intensive based on rules that would be refined, I defined a module that would specialize in synthesizing the messages. This module, called MessageSynthesizer, applied the mapping between UCM and XMI/SD concepts independently from the context it was used in. As MessageSynthesizer applied its mapping it knew whether a message was in a parallel block and hence could assign a label to it according to the mechanism described in 3.2.2 b).

3.2.4 Commonalities with UCM2MSC

As SD and MSC scenarios were visually similar, I found that UCM2XMI and UCM2MSC had many technical commonalities as well. Duplicating parts of UCM2MSC in order to use them for UCM2XMI would have been a bad software engineering decision. Finding an error in a common part to both applications (determining the instances for example) would necessitate the modification of two separate pieces of software. So there was a clear advantage and gain in software quality if duplication was avoided. However, avoiding duplication (a copy paste operation) and opting for reuse (effective sharing) had the disadvantage of time constraints.
It is considered as a software engineering principle that the advantages of reuse largely outweigh its disadvantages (the advantages of duplication). This principle held very well in my situation since by opting for reuse I was able to meet the deadline and produce software of higher quality.

3.3 Extending the project to UCMExporter

The need for reuse between UCM2MSC and UCM2XMI that we discussed in 3.2.4 enlarged the scope of my project. After discussing the new scope with Dr. Amyot, we agreed to evolve the project to building an open source tool that would enable the UCM-based transformations. I chose the name of UCMExporter for this tool (in parallel to UCMNav).

3.3.1 A Description of UCMExporter

The goal behind UCMExporter was to have a unique piece of software that would enable the UCM-based transformations. Software engineers would use UCMNav to generate XML scenario traversal descriptions, which we refer to from now on as *UCMNavXML*. Then, they would use UCMExporter to transform the UCMNavXML into TTCN, MSC, XMI/SD, or any other notation that we would support in the future. Figure 9 gives a black-box view of UCMExporter.

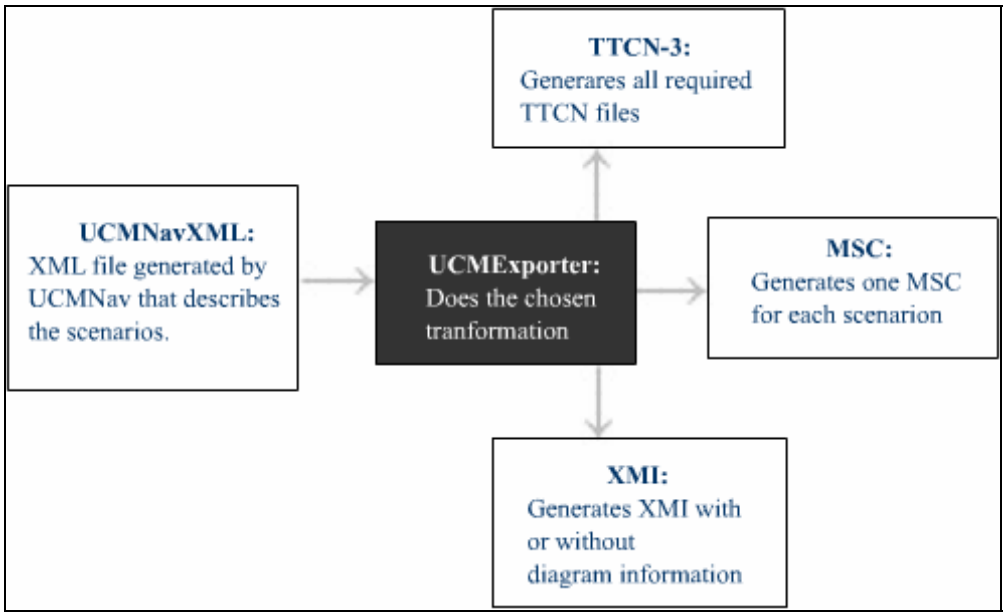

Figure 9 - Black-box view of UCMExporter

UCMExporter consisted of a command line application and a GUI (shown in Figure 10). I chose to implement it in Java in order to make it platform independent, but also because Java had well documented XML libraries (JDOM) and well known and capable XSL engines (XSLTC).

UCMNav was made open source, and similarly we wanted UCMExporter to be open source as well. The idea of having open source software was to encourage the involvement of other engineers from the academic and industrial sectors. Users of UCMExporter would have access to the source code and could hence review it, improve it, or make suggestions for us to do so. UCMExporter was hosted in the well-known open source software website SourceForge.net. SourceForge.net provided tools that were tailored to the needs of open-source software developers such as version control (CVS), backups, and web-hosting.

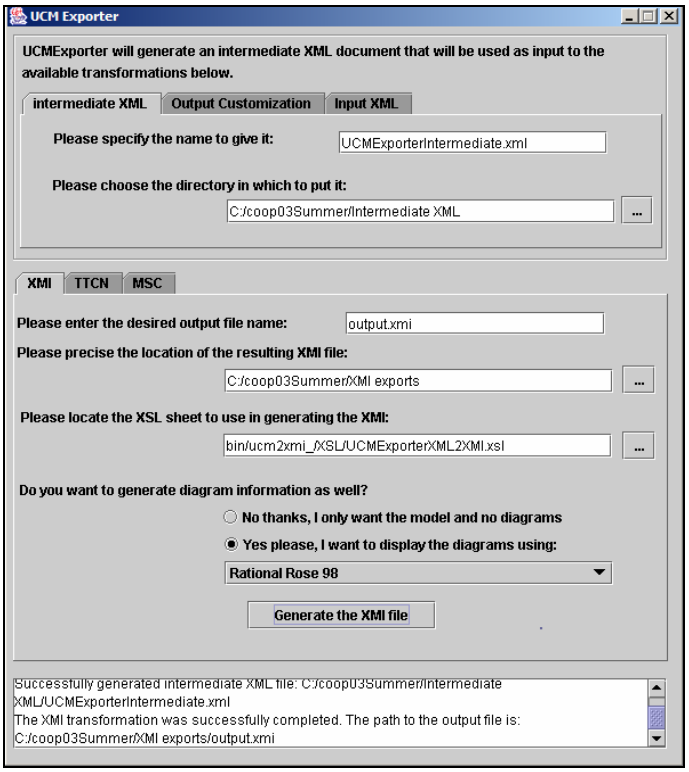

Figure 10 - Screenshot of UCMExporter

3.3.2 Integrating UCM2TTCN into UCMExporter

In order to allow the transformation of UCM scenarios into TTCN, UCMExporter needed to integrate the functionality of UCM2TTCN. The past work on UCM2TTCN consisted of one XSL sheet that did fairly simple transformations; hence integrating it was a simple matter. All that UCMExporter needed to do was to take the UCMNavXML (the input) and apply the UCM2TTCN XSL sheet on it.

3.3.3 Combining UCM2MSC and UCM2XMI: The Need for an Intermediate Step

The need to reuse the shared aspects of UCM2MSC and UCM2XMI resulted in introducing an intermediate step between the input (UCMNavXML) and the two applications. This intermediate step was responsible for generating the information that both UCM2MSC and UCM2XMI needed (such as instance names).

This intermediate step made the final transformations to MSC and XMI easier since all the needed information was ready to use. The intermediate step also improved code quality because it abstracted the complicated functionality and allowed for easier defect fixing, code maintenance, and evolution. Therefore, we expected that the transformations for future notations would be made easy as well.

3.4 Challenges and architectural choices for UCMExporter

As described in 3.3.3, I introduced an intermediate step between the UCMNavXML and the transformation applications. The intermediate step was packaged as *XMLInputTransform*. It became responsible then for modifying the UCMNavXML by adding more information to it and making it readily available for the later transformations. In order to benefit from the advantages of XML, we decided to make the result of XMLInputTransform in XML format. And in order to provide some backwards compatibility, and reuse the existing UCMNavXML DTD (Data Type Definition) we decided to extend it instead of defining a new one. We refer to the result of XMLInputTransform as *UCMExporterXML*.
Therefore, UCMNavXML and UCMExporterXML looked very similar but the latter had more readily available information than the former. Figure 11 and Figure 12 show a short example of a UCMNavXML and its corresponding UCMExporterXML respectively.

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE scenarios (View Source for full doctype...)>
- <scenarios date="Thu Aug 28 15:41:09 2003" ucm-file="SDLforum10.ucm" design-name="SDLforum10" ucm-design-
    version="117">
  - <group name="TeenLine" group-id="3">
    - <scenario name="TLnotActiveSuccess" scenario-definition-id="1">
      - <seq>
          <do hyperedge-id="0" name="req" type="Start" component-name="User" component-id="1" component-
            role="Orig" />
          <do hyperedge-id="50" name="start" type="Connect_Start" component-name="Agent" component-id="0"
            component-role="Orig" />
          <condition hyperedge-id="57" label="TeenLine" expression="bv4" />
          <do hyperedge-id="58" name="start" type="Connect_Start" component-name="Agent" component-id="0"
            component-role="Orig" />
          <do hyperedge-id="60" name="checkTime" type="Resp" component-name="Agent" component-id="0" component-
            role="Orig" />
          <condition hyperedge-id="62" label="[notActive]" expression="!bv1" />
          <do hyperedge-id="59" name="success" type="Connect_End" component-name="Agent" component-id="0"
            component-role="Orig" />
          <do hyperedge-id="49" name="snd-req" type="Resp" component-name="Agent" component-id="0" component-
            role="Orig" />
```

Figure 11 – A Sample UCMNavXML (truncated)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <scenarios date="Thu Aug 28 15:41:09 2003" ucm-file="SDLforum10.ucm" design-name="SDLforum10" ucm-design-
    version="117">
  - <components>
      <component name="User" id="C1" />
      <component name="Agent" id="C2" />
    </components>
  - <group name="TeenLine" group-id="3">
    - <scenario name="TLnotActiveSuccess" scenario-definition="">
      - <instances>
          <instance id="I1" name="User_Orig" component="User" role="Orig" desc="" />
          <instance id="I4" name="User_Term" component="User" role="Term" desc="" />
          <instance id="I2" name="Agent_Orig" component="Agent" role="Orig" desc="" />
          <instance id="I3" name="Agent_Term" component="Agent" role="Term" desc="" />
        </instances>
      - <seq para-label="">
          <do hyperedge-id="0" name="req" type="Start" component-name="User" component-id="1" component-
            role="Orig" />
          <do hyperedge-id="50" name="start" type="Connect_Start" component-name="Agent" component-id="0"
            component-role="Orig" />
          <message id="M2" name="req" source-id="I1" destination-id="I2" is-task="false" is-timer="false" timer-property=""
            last-ref="" desc="" para-label="" is-connector="false" connector-type="" />
          <condition hyperedge-id="57" label="TeenLine" expression="bv4" instance="I2" />
          <do hyperedge-id="58" name="start" type="Connect_Start" component-name="Agent" component-id="0"
            component-role="Orig" />
          <do hyperedge-id="60" name="checkTime" type="Resp" component-name="Agent" component-id="0" component-
            role="Orig" />
          <message id="M3" name="checkTime" source-id="I2" destination-id="I2" is-task="true" is-timer="false" timer-
            property="" last-ref="" desc="" para-label="" is-connector="false" connector-type="" />
          <condition hyperedge-id="62" label="[notActive]" expression="!bv1" instance="I2" />
          <do hyperedge-id="59" name="success" type="Connect_End" component-name="Agent" component-id="0"
            component-role="Orig" />
          <do hyperedge-id="49" name="snd-req" type="Resp" component-name="Agent" component-id="0" component-
            role="Orig" />
          <message id="M4" name="snd_req" source-id="I2" destination-id="I2" is-task="true" is-timer="false" timer-
```

Figure 12 – The UCMExporterXML corresponding to the UCMNavXML in Figure 11

Since UCMExporter was to integrate the functionality of all existing transformations, it had to cope with their problems as well. And because we wanted it to be a complete tool that could be extended for future transformations, it had some challenges of its own.

Let's look at the main problems and challenges that UCMExporter faced and how I approached them.

3.4.1 The mapping between UCM2MSC and UCM2XMI

In order to define the mapping between both UCM to MSC and SD concepts, we reused and augmented the MessageSynthesizer described in 3.2.3. The sole responsibility of MessageSynthesizer was to detect the messages that needed to be shown in any of the supported notations. By using the attributes of a message, the later transformations knew if that message was suitable for them to show or not. For example, a message that had a timer flag set would be interpreted differently by UCM2MSC and UCM2XMI.

This separation of responsibilities, made the potential support of another notation easier. As I explained in 3.3.3, UCM2MSC and UCM2XMI became easier to implement. In fact UCM2MSC and the first step of UCM2XMI (described in 3.2.3) became both straight forward. That was due to the choice of using XSL for implementing the transformations.

Even though UCM2MSC, UCM2XMI, and any other potential transformation would receive the same UCMExporterXML input, they would only see the parts that were specified in their XSL sheet and would discard the rest. This would make adding new information to the UCMExporterXML in order to satisfy the needs of a new transformation invisible to the existing ones. In fact that was experienced as I had to change the intermediate step to add information that was needed by UCM2MSC but not UCM2XMI in order to improve the former. And that caused no side-effects to the latter.

3.4.2 Parallelism in SD

As mentioned in 3.4.1, I augmented MessageSynthesizer which was already applying the parallelism labeling mechanism described in 3.2.2 b). Therefore, when the intermediate step added a message for UCMExporterXML it defined a parallelism label attribute that would hold that information (e.g. para-lable=”p1.s2”). And as I explained, the use of XSL for the transformations made this piece of SD-specific information visible to UCM2XMI but not to UCM2MSC.
Hence the decision to opt for reuse instead of duplication helped applying the same implementation of the solution to this problem.

3.4.3 XMI/SD Layout Problem

In 3.2.3 we looked at the architectural choices for the initial UCM2XMI. The decision to split UCM2XMI into two steps proved fruitful. The changes to UCM2XMI described in 3.4.1 concerned only the first step and consisted of writing a straight forward XSL sheet to produce standardized XMI. The second step required no modifications since it only depended on a standardized XMI and added tool-dependent diagram information to it regardless of how the input was generated.
The layout problem was a major one and the fact that I was able to reuse the second step proved the ease of maintainability and evolution of the design.

3.4.4 Connectors for Causality and Names:

After the problems mentioned above were tackled and successfully solved, we moved to an issue that concerned both MSC and SD notations and that was the issue of causality. The problem of causality consisted of making sure that the execution of the scenario was continuous, i.e. that there were no sudden unconnected jumps from one instance to another. Figure 13 shows a sample causality problem.

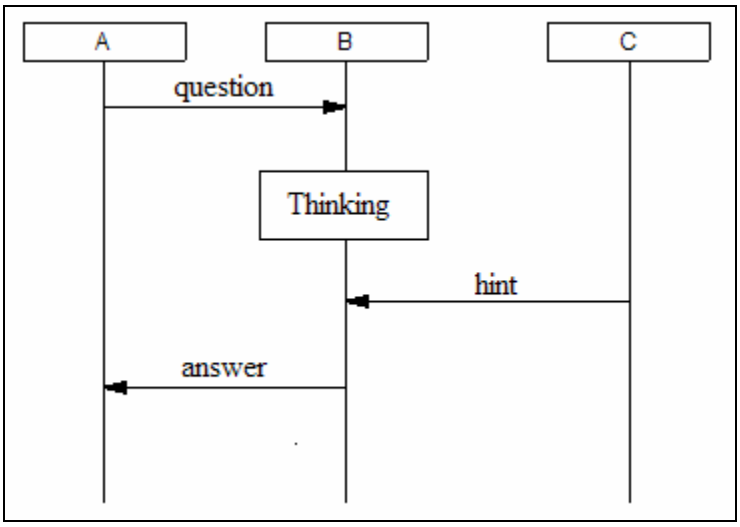

Figure 13 - An example of causality problem

As we can see from the figure above, the scenario started when A sent a question to B and then B started thinking about an answer. Suddenly, C gave B a hint and then B answered. To contrast this with the real-life, imagine A was a teacher and when she asked her student B a question, B's classmate C gave him a hint. While this sounds normal in a classroom setting, it is implicitly said that B heard the question and C did too (or maybe B asked C for a hint after some thought). This shows that it would not be normal that C gives B a hint if C did not hear the question (received a message from A) or if B did not ask for one (B sent message to C). Therefore, it would be more correct to preserve the causality by having a message sent to C (either form A or B depending on the scenario we choose). Figure 14 shows how the scenario should have been displayed in MSC.

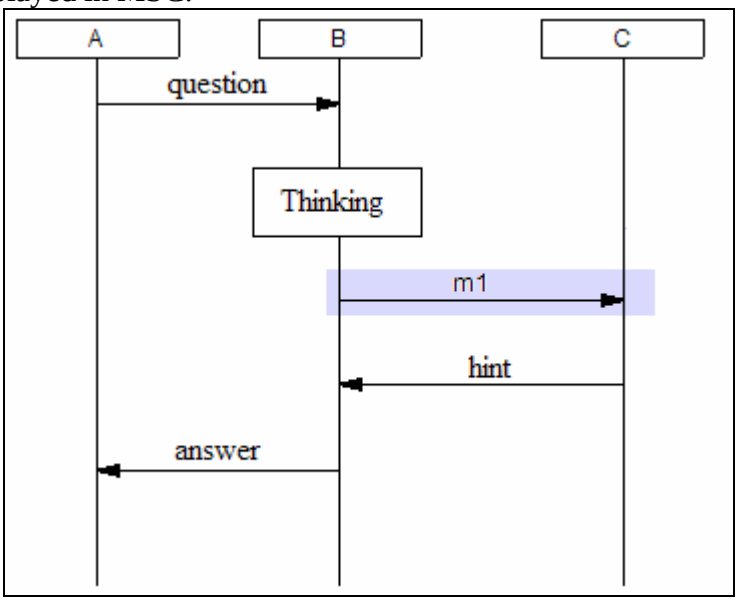

Figure 14 - Solution to Causality problem

UCM models are used to think of a system at a very high level of abstraction. They don't express the causality explicitly because it is a lower level issue. The UCM models allow the software engineer to express things implicitly about the communication between components (as the classroom example I gave above). Therefore, UCMExporter had to account for those missing (implicit) communications between the components (or instances) of a system.

After the need of messages to keep the causality was established, we moved to defining the rules to detecting them. We referred to that type of messages as *connectors.*

The general rule consisted of detecting transitions of execution from one instance to another that lack communication (e.g. sending a request). While detecting this visually was straight forward, we needed to detect it by analyzing the UCMNavXML.

I introduced a module in the intermediate step that would be responsible for the generation of the connectors. The modularized architecture allowed for such integration of new modules with minimal changes to the rest of UCMExporter. We later implemented the rules as they became available. We found two sets of rules. A set of rules that dealt with instances interacting in sequence, and another set of rules for parallel communication. The first set of rules was straight forward compared to the second.

Detecting connectors for sequential communication consisted of remembering the current instance and the previous instance as we parsed the UCMNavXML. If we detected that the previous and current instances were different, and that there was no explicit communication between them then we generated a connector between the two.

However, detecting connectors for parallel communication necessitated a larger understanding of what was happening in the UCMNavXML. Dr. Amyot and I identified four rules that were based on the following information:

1. The last instance (or instances) before entering a parallel block.
2. The instances that first execute inside the parallel block
3. The instances that last execute inside the parallel block
4. The first instance (or instances) after exiting the parallel block.

Based on the information above we could detect the need for connectors and determine the sender instance and the receiver for each connector. Figure 15 shows one of the rules for parallel related messages. It defines a mapping between a parallel structure in UCM and the corresponding SD. The detected connectors are shown as m1 and m2. For a description of the UCM notation for parallelism and the rest of the parallel connector rules, please refer to Appendix A: Mapping

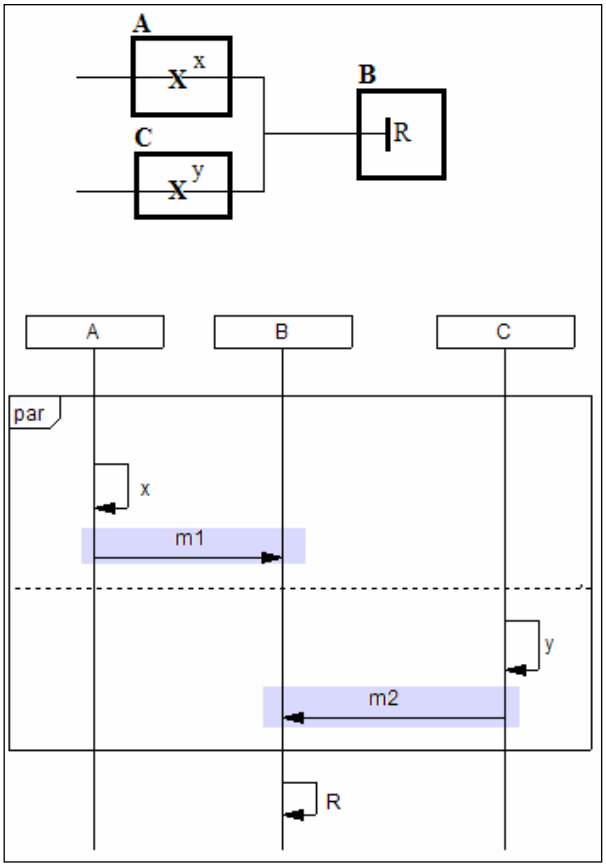

Figure 15 - Sample Parallel connector rule (Rule 1)

The names that were given to the connectors (e.g. m1, m2) were not descriptive enough. While descriptive names would help software engineers better understand the scenario, they were important for the synthesizing of SDL models form a set of MSC files.

Yong He, one of Dr. Amyot's Phd students, pointed out the need for having consistent names for the connectors. This meant that a connector called m1 should have the same meaning in different scenarios of the same model.

Therefore, instead of incrementally generating connector names (m1, m2, m3, etc…) I defined a simple method to give context dependent names. The method consisted by looking at what the message connector's sender was doing before sending it, and what the receiver would do after receiving it. If the sender was doing operation X, and the receiver would do operation Y, then the connector would be named "did_X_do_Y". In the classroom example given above, we would change the message m1 in Figure 14 to "did_Thinking_do_hint".

The method I defined gave a context dependent description, and solved part of the problem by making sure that if two connectors had the same name than they must mean the same thing with respect to what was happening immediately before and what would happen immediately after. The method was not suitable for giving names by looking at the entire context because they would be too long. Therefore, that was a partial solution to answer a particular need for synthesizing SDL models from a set of MSC files.

3.4.5 Allowing for Customization

One major requirement for UCMExporter was to allow for results customization, i.e. allow the user to override an existing or specify a new transformation rule. The main application of the customization would be the definition of specific communication protocols between participating instances.

In order to add this functionality I added a customization module that could execute a user specified XSL sheet. In the XSL sheet, the user would write the new rules to apply in order to modify the final result. Normally a user would view the default output of UCMExporter and then write one or many XSL sheets to test different communication patterns. The user could define a communication protocol between a specific pair of instances, or introduce a group of instances that would act as brokers. Please refer to Appendix B: Case Study for a sample customization.
The modularity of UCMExporter made plugging-in a new customization module easy. The module was inserted at the end of the intermediate step. Hence the results of the intermediate step (UCMExporterXML) contained the customized information and therefore the later transformations to MSC or XMI/SD needed no modifications.

3.5 Validating the Results:

After the implementation of the transformation rules, and after I verified that they were working properly (no defects), I needed to validate their results to make sure that what they were doing was correct.
In order to validate the transformation process, Yong He gave me 15 UCM scenarios and their manually generated corresponding MSC files. The 15 scenarios contained the different aspects of scenarios such as parallelism and timers. The rules would be considered valid, if the output of each UCM corresponded to its manually generated MSC.
We successfully reproduced the given MSC files from the UCM scenarios. Each generated XMI/SD was easy to check against the expected MSC. However, I expected the need for future rule corrections or refinements and used the modularized approach in order to localize the required changes as much as possible.

3.6 Expected Improvements:

There were improvements that I wanted to make but could not given the time constraints. One improvement was the creation of a web interface for UCMExporter. The web interface would allow users to use the internet to enter their UCM scenarios and get the corresponding TTCN, MSC or XMI/SD files. This would eliminate the need for packaging and installing the application.
Another improvement that I attempted was the definition of a module that would generate all the possible interleavings in a parallel block. Implementing it would allow an automated SDL synthesizer to synthesize better models from a set of MSC files. However, that was a complex matter that I could not tackle within the given time frame.
Also, the code could also be improved by simplifying some parts and adding some error checking. In order to allow for such improvements, I commented my code abundantly and used the Javadoc program to put the code documentation online for future developers.

4. Conclusion

I was very satisfied with my co-op term with Dr. Amyot during the summer of 2003. I enjoyed completing my project and benefited at the professional, personal, and technical levels.

At the professional level, I improved my ability to work independently under minimal supervision. The weekly group meetings and my regular discussions with Dr. Amyot helped me improve my presentation and communication skills in both French and English. I also had the chance to do requirement gathering and improve my ability to identify the rules and requirements.

At the personal level, I enjoyed working in the new SITE building of the University of Ottawa. I also enjoyed meeting PhD and Master's students of the faculty and asking them for career recommendations and advices. Moreover, because of the time flexibility I could organize my schedule to do more extracurricular activities.

The technical level was certainly the level at which I benefited the most. I used some technologies for the first time (such as XML, XMI, and XSL), and had the chance to review some others (such as UML and MSC). I became more familiar with the Eclipse Java Development platform since I used it to write, test and execute around six thousand lines of Java code for this project. I also used the Xselerator tool for the first time for writing, testing and executing 1200 lines of XSL code.

The job description explained very well the purpose of the project and referred to the technologies that would be used. And during the interview, Dr. Amyot explained to me the possible problems and challenges that could be faced and gave me the big picture of his work.

Dr. Amyot considers his co-op students as full members of his group and they are informed of other members' work and progress. The projects that Dr. Amyot assigns to his co-op students are important ones in terms of scope and future integration with other projects.

5. References

[1] F. Bordeleau, D. Cameron: *Relationship between MSC and UCM*

[2] D. Amyot, Y. He, D. Amyot, Y. He, X. He: *Generating Scenarios from Use Case Map Specifications*

[3] http://www.geocities.com/SiliconValley/Network/1582/uml-example.htm#toc, Anuar Musa: *Unified Modeling Language by Example*

[4] Y. He, D. Amyot, A. Williams: *Synthesizing SDL from Use Case Maps: An Experiment*

[5] http://www.usecasemaps.org : *The Use Case Maps Web Page*

[6] http://www.oasis-open.org/cover/xmi.html : *Cover Pages: XMI*

[7] http://www.w3.org/XML/: *EXtensible Markup Language*

[8] http://www.etsi.org/frameset/home.htm?/ptcc/ptccttcn3.htm: *ETSI Telecom Standards*

6. Appendix A: Mapping

6.1 UCM Concepts to SDL and MSC Concepts

In this section I give a summary of the main three concepts of UCM that are translated into MSC and SD concepts, and they are:

1. The concept of *responsibility* in UCM is translated as an *action* in MSC and a *self message* in SD.
2. The concept of *Timer_Set* is translated as a *Timer Set* in MSC which shows that a timer was started. In SD this will mark the start of a self message.
3. The concept of *Timer_Reset* is translated as a *Timer Reset* in MSC which shows that a timer was reset. This will mark the end of a self message in SD.

The mapping for other UCM concepts such as *Start*, *Connect_Start*, and *Connect_End* is explained in the UCMExporter documentation.

6.2 Parallel Connector Rules

As mentioned in 3.4.4, we identified four rules to detect the connector messages in parallel blocks. The first rule was show in Figure 15, but I put it here again and explain it in more detail.
In each rule we give a Use Case Map that shows a possible parallel behavior and below it is the corresponding MSC.

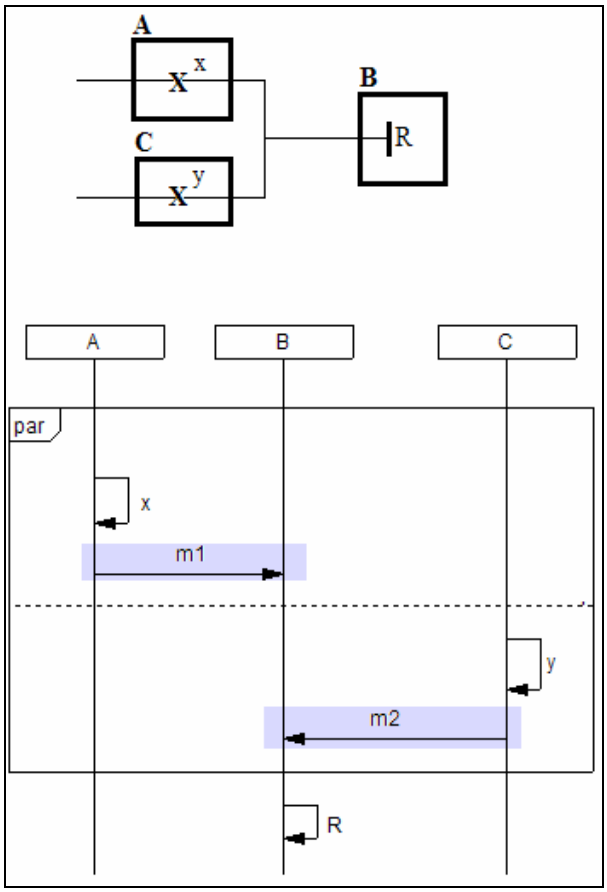

Figure 16 – Parallel Connector Rule 1

The Use Case Map expresses the following:

- There are two actions x and y that are in parallel
- The action x is happening in component A and y is happening in component C
- When both actions x and y are finished, we go to a third component B and execute action R.

The parallel connector m1 shown above expresses the implicit communication between A and B. The parallel connector m2 expresses the implicit communication between C and B. When receives both messages, it will know that A and C finished their respective actions. Therefore B can go ahead and execute R.

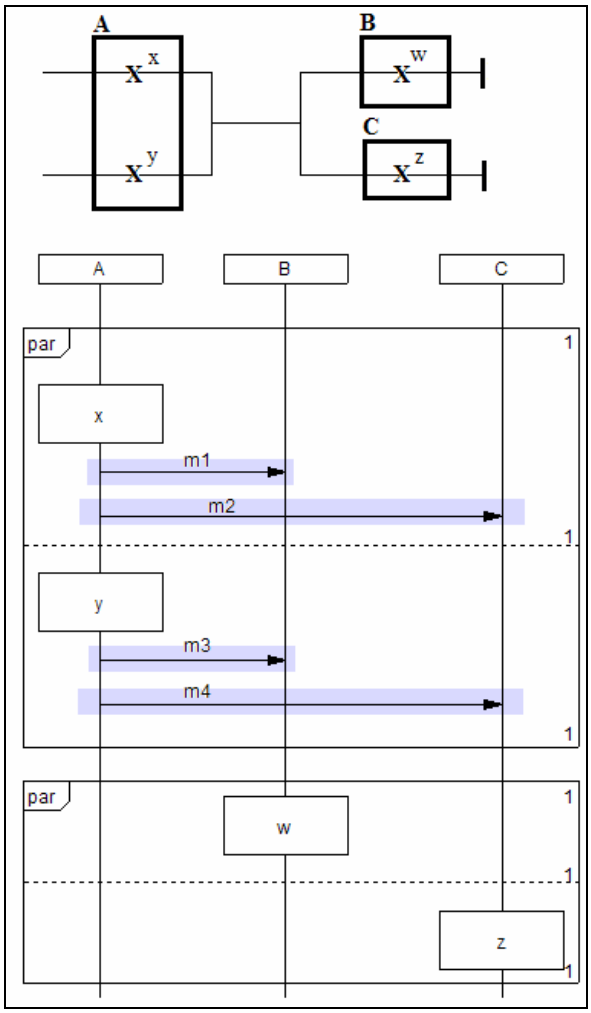

Figure 17 - Parallel Connector Rule 2

The Use Case Map expresses the following:

- We have three components: A, B, and C
- We start by having two parallel actions (x and y) inside A
- Once x and y are completed we start two other parallel actions (w, z).
- The parallel actions w and z are in separate components: B and C respectively

From the UCM we can see that actions w and z should not start until x and y are finished. Therefore there should be a message that will signal the end of x and y. Since there are two components when we leave the parallel block we need A to send two messages for B and two for C.
Connector messages m1 and m3 tell B that actions x and y are done. Connector messages m2 and m4 tell C that x and y are done. Actions w and z in B and C respectively should only start when those messages are received.

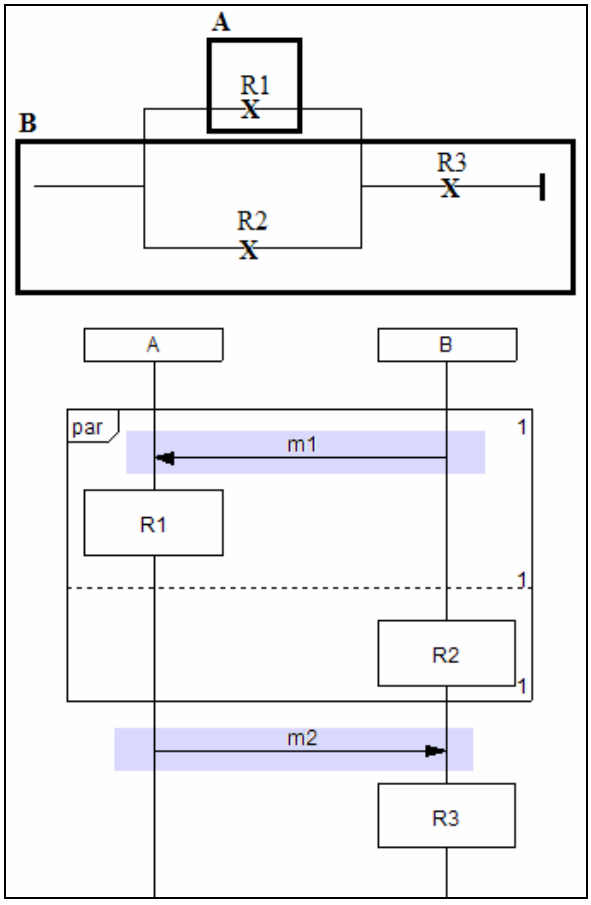

Figure 18 - Parallel Connector Rule 3

The Use Case Map expresses the following:

- We start with two parallel actions R1 and R2.
- R1 is happening inside A, and R2 inside B.
- When both R1 and R2 are done, we execute action R3 which is in B.

The message m1 is shown in this MSC to trigger the parallel behavior. It could be removed without loosing the causality. However m2 is needed to show that R3 will only execute once the parallel execution is complete.

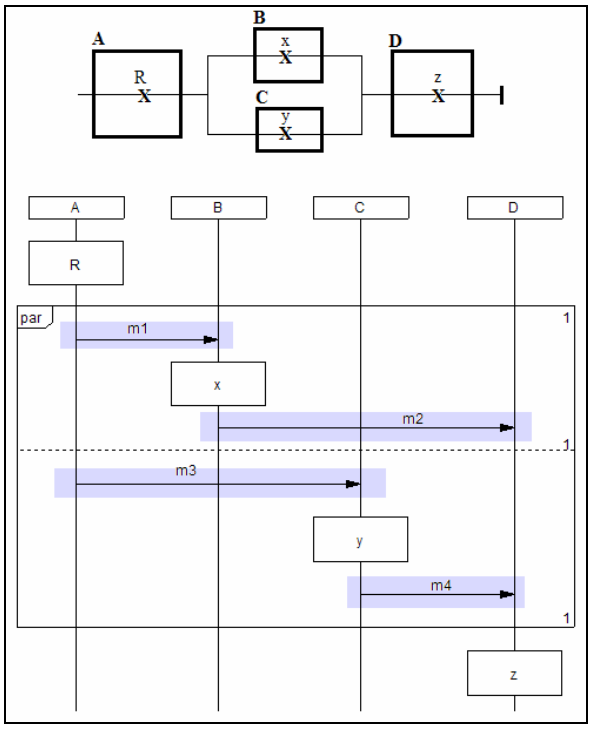

Figure 19 - Parallel Connector Rule 4

The Use Case Map used in this rule expresses the following situation:

- The execution starts with action R in component A
- We then have a parallel execution of actions x and y in components B and C respectively
- When x and y are done, we go inside a fourth component D and execute action z.

We start the execution with action R in component A. In order to have the parallel action x and y start, A sends m1 and m3 to B and C respectively. Once the parallel execution of x and y is finished, B and C inform D that it can start z by sending it m2 and m4.

7. Appendix B: Case Study

This case study serves the purpose of illustrating how UCMExporter would be typically used. The case study is broken into several steps, each being commented. This case study shows how to generate XMI/SD only, but the same procedure holds for MSC and a more straight forward one for TTCN.

7.1 The Use Case Map

A software engineer wishing to use UCMExporter must define or use a predefined UCM. To define a new UCM or to edit an existing one, the software engineer would use the UCMNav tool.
In this case study, we will use the UCM shown in Figure 1 and transform the scenario highlighted in Figure 2. The user should generate the UCMNavXML by selecting the scenario name and then hit the "To XML" button shown at the bottom.

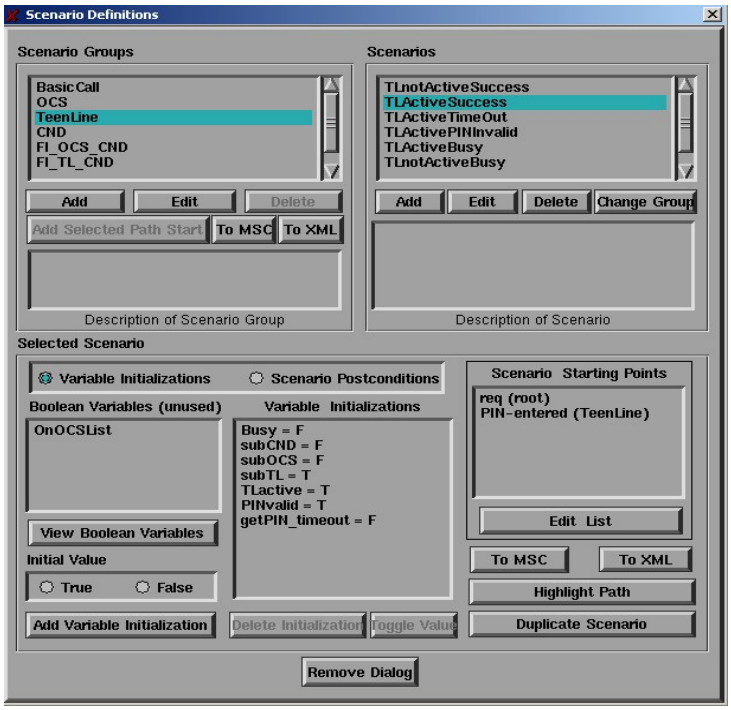

Figure 20 – Generating UCMNavXML with UCMNav

7.2 UCMExporter to Generate the XMI/SD

The software engineer would then launch UCMExporter and choose the UCMNavXML generated in the previous step as the input. The user could choose the three types of output available TTCN, XMI, or MSC. In our case study, we will choose XMI.

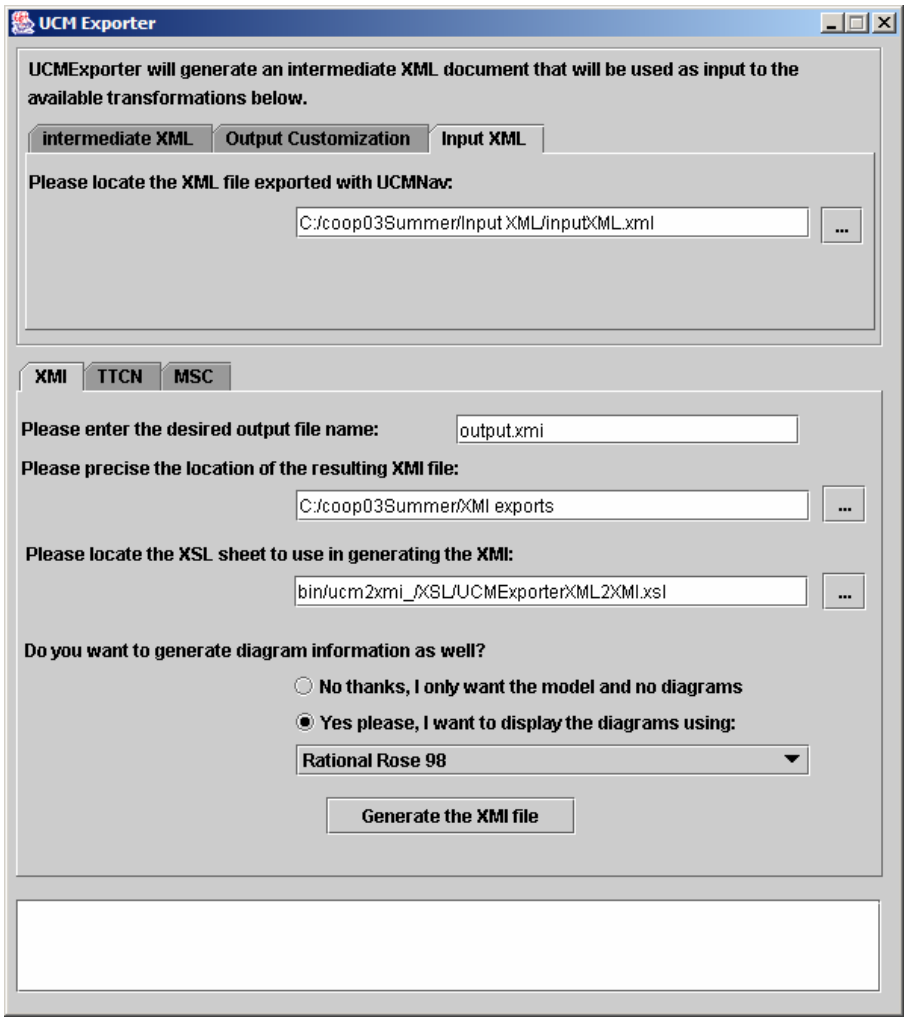

Figure 21 – Generating XMI/SD using UCMExporter

The user could choose to generate only standardized XMI, i.e. XMI with no tool specific diagram information. This would be useful if some tool would do the XMI auto-layout in the future. However since no tool was known to do that, we provided the option to target a specific UML tool. UCMExporter supported Rational Rose 98 only (Rational Rose for short); however it was designed to be extended to target other tools as well. Once ready, the user will hit the "Generate the XMI file" button. The XMI file would be given the name and would be put in the directory specified in the GUI.

7.3 Viewing the XMI/SD

The user could visualize the result using Rational Rose 98 as specified in the previous step. As I mentioned in 3.2.2 b), Rational Rose used a plug-in developed by Unisys for its XMI importing and exporting.

The result that Rational Rose would show was given earlier in Figure 8. The resulting SD had two connector messages whose names were customized to did_snd_req_do_ringTreatment and did_ringingTreatment_do_fwd_sig respectively using the mechanism explained in 3.4.4.

7.4 Customizing the result

Now that the user saw the output, they could decide to implement a specific communication protocol when some conditions of their choice hold. A condition could be:

- Use the protocol between some specific instances
- Use the protocol for some specific type of messages
- Use the protocol for some specific messages (given their IDs)

In this example, we decided to customize all connector messages using a dummy protocol named “Say It Again Please”. This protocol defines a communication pattern for all connector messages that consists of having the receiver request a repetition of the message.
The protocol would be defined in an XSL sheet which would define the new protocol rule. The user would then tell UCMExporter where to find the customization XSL file.
I wrote such an XSL sheet, and below is the output after the customization took place. The figure blow shows the resulting XMI/SD. It shows that the protocol was applied to the two connector messages and not the rest as the rules specified.

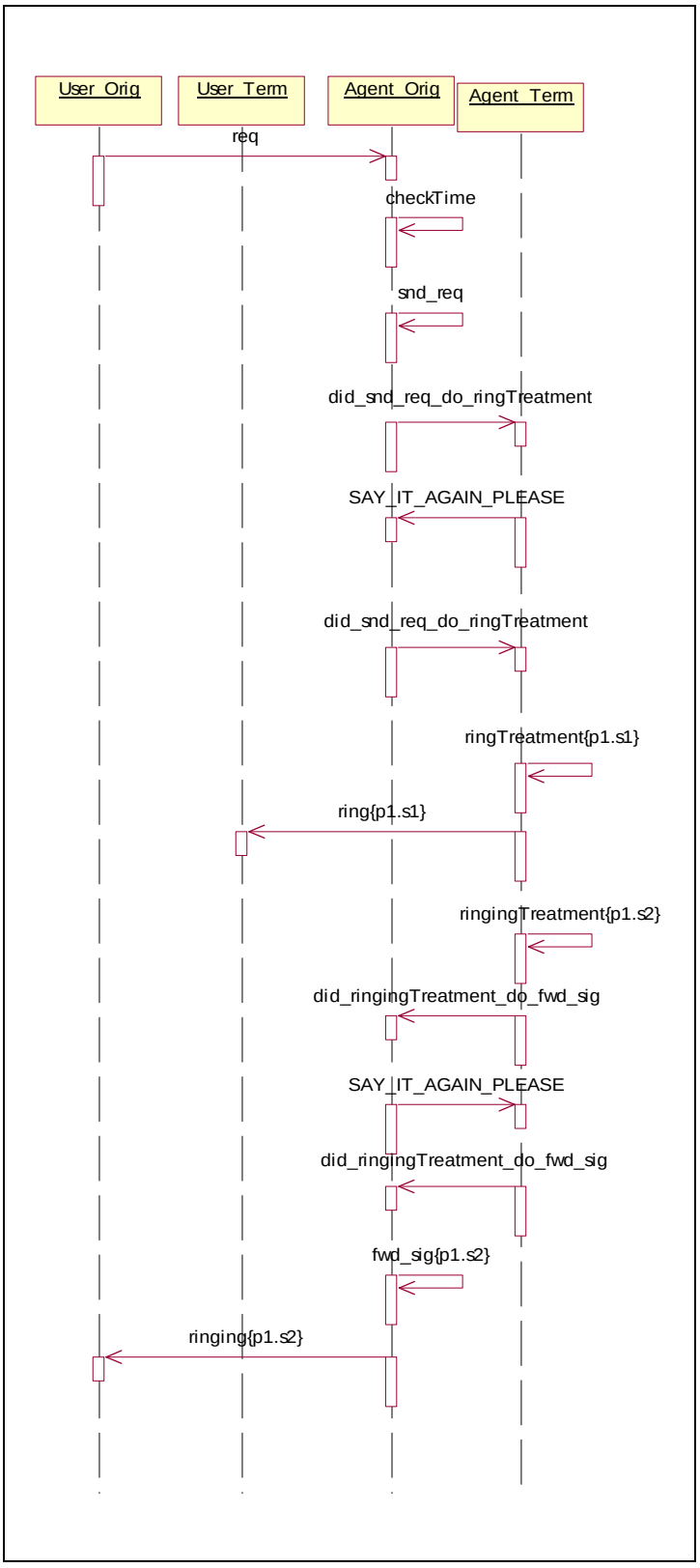

Figure 22 – Customized XMI/SD result

8. Appendix C: Full picture of UCMExporter

Below is the overall picture of UCMExporter. It shows the steps for the supported transformation from UCMNavXML to TTCN, MSC or XMI/SD. Some notes and comments are given as well.

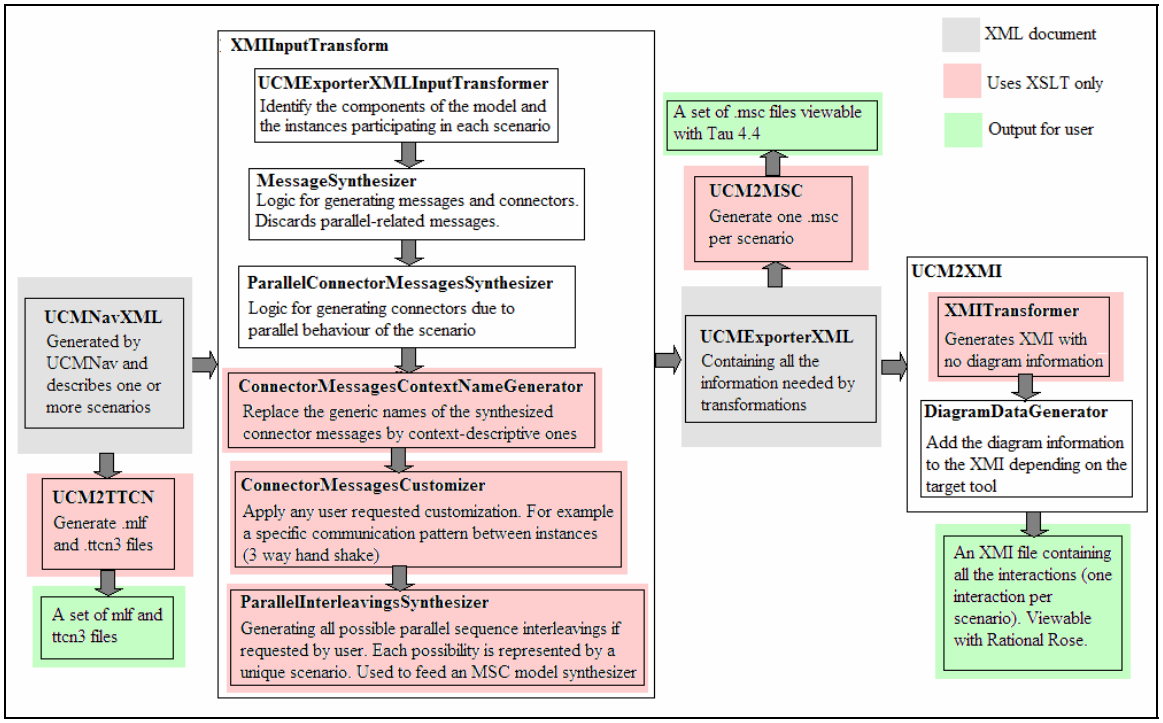

Figure 23 - Full picture of UCMExporter

Here is a summary of the transformation process. We generate UCMNavXML for one or more scenarios using the UCMNav. If we want to generate TTCN output, we can directly call the UCM2TTCN application.
However, if we want to generate MSC or XMI/SD we have to go through the intermediate step. The result of the intermediate step will be UCMExporterXML that has all the information that the following transformations will need.

The figure above shows also the internal modules of the intermediate step that were packaged as XMLInputTransform. The modular architecture allowed for inserting modules as they were made ready and made error localization (debugging) and correction much easier.