From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language

Igor S. Sales - isales@site.uottawa.ca
Robert L. Probert - bob@site.uottawa.ca
School of Information Technology and Engineering - S.I.T.E.

Abstract

The Telecommunications industry's demand has grown immensely for cost-effective methods for software validation & verification of mobile products. Semi-formal methods such as Use Case Maps and formal methods such as SDL are starting to be widely used. In this paper we show how to integrate these two techniques in terms of quickly validating high level designs in SDL against desired abstract behaviours in UCM notation.

Keywords: Use Case Maps, SDL, Mobile Systems, Telecommunications Software Engineering

1. Introduction

Today's customers of mobile services is an ever-faster growing market. Together with the legacy of current telecommunications systems, new services and features are being created, designed and sold all over the world. However, these features are more and more highly dependent on software and more and more complex. To currently design such features is not an easy task. Quality assurance, security, and time to market are all critical for these products. Designers require effective techniques and extensive design tools to achieve high levels of quality and security without sacrificing time to market. Some corporations which develop mobile systems and standards are supporting their designers in the use of formal techniques [1,2], such as the ones presented here, namely, Use Case Maps (UCM) [3] and the Specification and Description Language, SDL [4,5]. In this paper, we introduce a method for rapidly converting use case maps into SDL designs. This mapping is done by a designer using our rationale for correctly choosing which SDL structures to derive from UCMs. In Section 2, we introduce both notations, and in section 3 we give some rules for capturing and drawing UCMs to represent functional requirements. Then, in section 4 we give the UCM to SDL mapping process. Section 5 contains a case study, namely a high-level design of a Wireless Mobile Station. Finally, in section 6 conclusions and suggestions for further research are given.

2. Notations

Two notations are required for the full understanding of this work, Use Case Maps, UCMs and the Specification and Description Language, SDL. These two semi-formal and formal notations are both highly graphical. Use Case Maps are used in the early stages of design when Use Cases [6] and Use Case Diagrams [7] are being used to capture and refine the system requirements. The next two subsections will familiarize the reader with such notations. Figure 1 illustrates this paper's contribution.

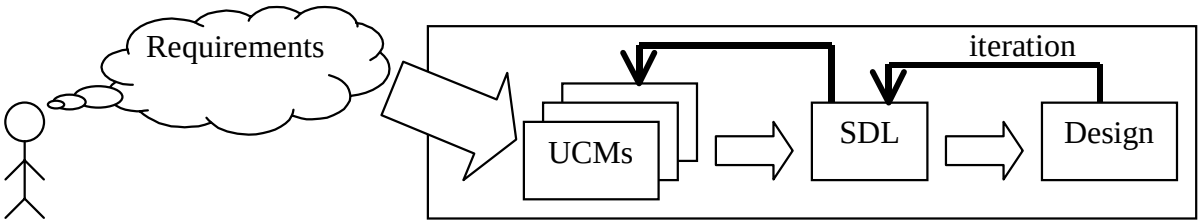

Figure 1 – From Use Case Maps to SDL

2.1. Use Case Maps - UCMs

A *Use Case Map*, *UCM* for short, is a means of describing the causal relationships among system events over time. A simple use case map consists of a *start point*, a *path*, and an *end point*. It is basically used to describe sequences of events, as in use cases [8], or in natural language descriptions of functional requirements. Figure 2 shows a very simple Use Case Map. A black circle indicates the *start point*. The *path* is drawn as a line, and the *endpoint* as a bar perpendicular to the thread. Figure 2 also includes the concept of a *stub*. One can think of a stub as being a sub-UCM. When we follow an UCM and find a stub we fall into the stub, finding another use case map. A responsibility, indicated by an "X" along the time thread, is an action. When time comes to that point that action must be executed.

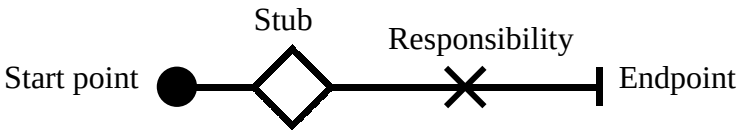

Figure 2 – A sample use case map

One of the drawbacks of writing English language use cases is the difficulty of representing the possible different outcomes and the introduction of ambiguities. With use case maps, this job is done easily using forks. Figure 3 shows the different possible constructions in this language. The dotted lines indicate that there is more unspecified information in the indicated direction. In Figure 3a we can see that a responsibility is required before an *or-fork* since it is the last action taken before deciding which way to follow. *And-forks*, shown in Figure 3 (c) and (d), provide concurrency. The *and-fork* splits into two or more concurrent paths, and the *and-join* synchronizes two or more paths.

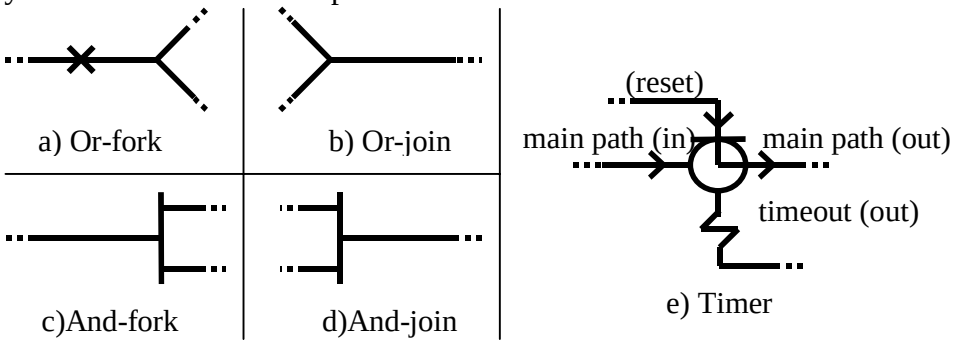

Figure 3 - Different Use Case Map Constructions

Another interesting construct is the *timer*, very important for real-time embedded systems[9]. Figure 3 (e) shows how this construct works. For a *timer*, there is the main path that follows

the path in the middle, and the alternative path going out from the bottom of the clock icon representing a timeout action. The *endpoint* that ends at the top of the clock indicates the path resetting the timer. Arrows were drawn here only for clarity purposes; they are optional in UCMs.

Finally there are *bound* UCMs and *unbound* UCMs. The meaning of *bound* is *to be allocated to a component, or to an architectural entity* [10]. Figure 4 below shows an example of an architecture for a Mobile Station. Components are modules of the system under design.

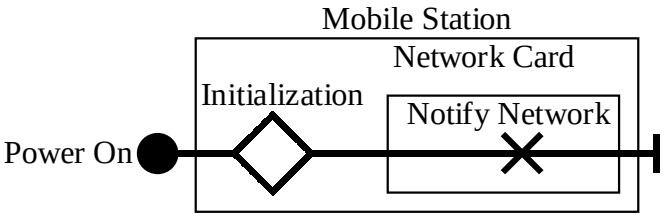

Figure 4 - A sample Mobile Station Architecture

The *"Power On"* label next to the start point indicates an event coming from the environment. This is the triggering event of the map above. *"Mobile Station"* and *"Network Card"* are two components of this system. *"Network Card"* is a sub-component *of "Mobile Station"*. The responsibility *"Notify Network"* has to occur inside the component named *"Network Card"* according to this design. When there is a polygon surrounding a UCM we say it is *bound*. In this work we will always refer to bound use case maps.

2.2. Specification and Description Language - SDL

An SDL specification consists of both *architecture* and *behaviour*. Common architectural components in SDL are referred to as *systems*, *blocks*, *processes*, and *channels*. Behavioural constructs in SDL specify the behaviour of processes. These are expressed in terms of SDL flowcharts, which are equivalent to Extended Finite State Machines (EFSM) [11].

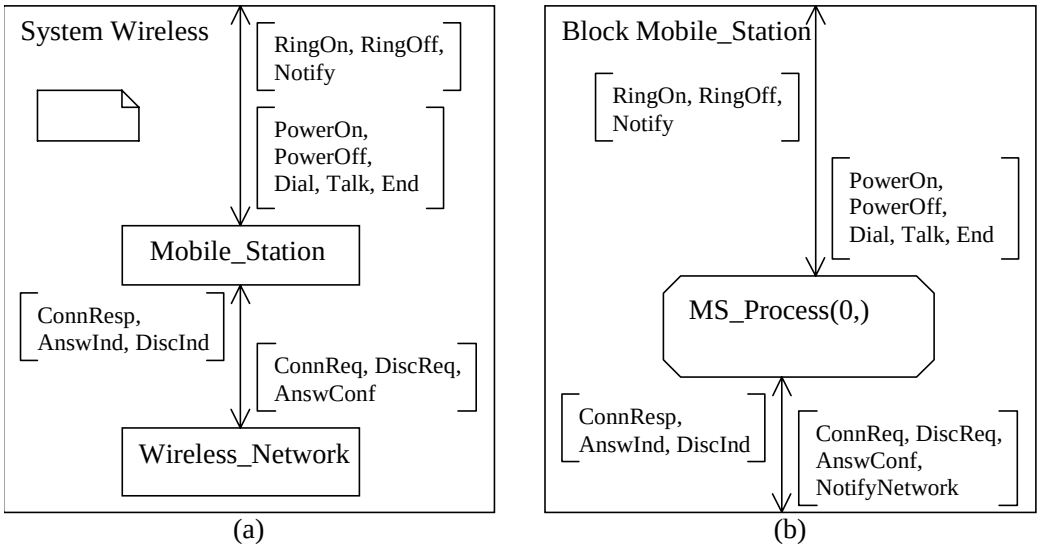

Figure 5 - A sample SDL architectural system

In Figure 5 details of an SDL system architecture for a mobile wireless protocol are shown. On the left side, in (5a), is shown the architectural decomposition of the system into

components or *blocks*. In 5(b) the architectural details of the block "Mobile_Station" are shown. The system architecture shows the interaction between blocks, and the block architecture shows the interaction between processes. The arrows connecting the system's environment to blocks and the block's environment to processes are called *channels* or *signal routes*. The channels have associated a set of *signals* with each side. *Signals* are triggers that stimulate the behaviour of the processes. These signals must all be declared inside the box with the document symbol (System definition block, Figure 5a). They were suppressed in that figure as irrelevant to our discussion. Inside each *process chart* the behaviour of that process is defined as a set of EFSM transitions (<current state, event, condition, output actions, next state>).

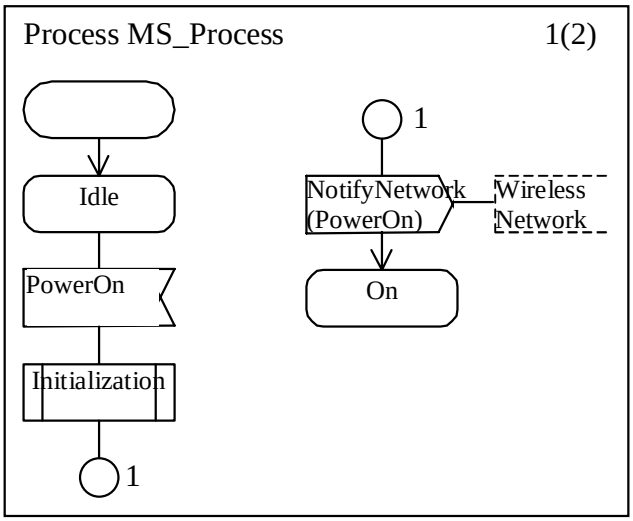

Figure 6 - A sample SDL process chart

The behaviour described in Figure 6 is based on the use case map from Figure 4, except that there is no Network Card entity in this SDL model. The SDL behaviour is described in a flowchart-like format. The first symbol of the process, right below the process name describes the starting point of the starting transition of the process. It immediately goes to the state "Idle". This is a way of indicating the initial state of the process. Upon receiving the input event "PowerOn" in the state idle, the process calls the procedure Initialization, and then sends a message NotifyNetwork with the parameter PowerOn to the Wireless_Network block. Moreover, it moves to the next state labelled "On". This state has its transitions defined in another process chart labelled MS_Process 2(2) (chart 2 of 2) not shown here. Each state may have several different inputs, and/or conditions leading to different actions and therefore different states. The circles indicate continuation points. It is a simple feature for drawing long diagrams that do not fit within the entire height of the chart.

3. Capturing Functional Requirements with UCMs

Functional Requirements are those behaviours that affect the user directly. These requirements are often represented by sequences of actions/reactions called *scenarios*. It is not an easy task to capture such requirements in a EFSM model [12]. However, it becomes easier if we focus on the signals exchanged between the user and the system (the *user interface* or *system environment boundary*). Figure 7 shows a typical extraction of functional requirement signals using *use case diagrams*[13] (7a) and *use case maps*[14] (7b). On the left side (Figure 7a) the user can turn on the mobile station, or place a call or receive a call. On the right side

(b), a UCM view of the same information is captured, this time in terms of actual observable actions to and from the system (in this case a Mobile Station).

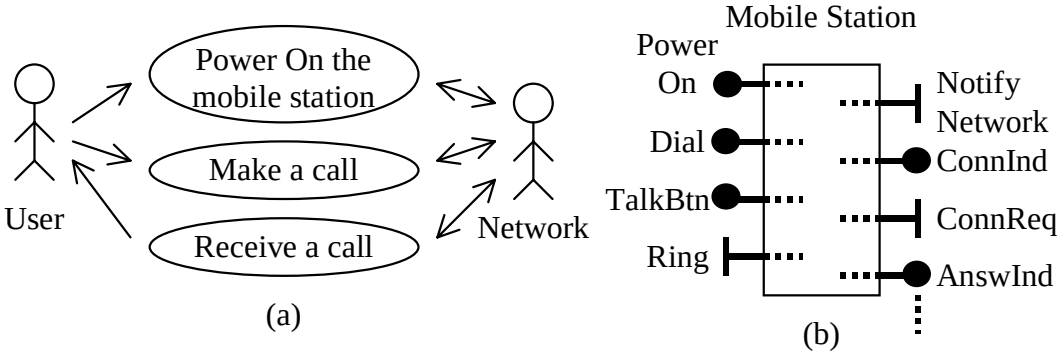

Figure 7 - Use Case Diagram and Use Case Maps

From the use case, “Turn on the mobile station” requires an action which is, for instance, a button named “PowerOn”. In order to “Make a call”, the user needs to “Dial” the destination number and hit the “Talk” button. To “Receive a Call” the user needs to be notified that someone is trying to call him/her. This is done through, for instance and audible “Ring”. To accept the call the user should simply press the “Talk” button.

Similarly, with the network behaving as a user, it is possible to acquire the messages on the network side. Note that we assume the convention that start points indicate signals from an actor to the system, and signals attached to an endpoint are used to refer to incoming signals going to the actor. Figure 7 does not show all messages for the network side, instead, showing a vertical reticence symbol to indicate that there exists more messages to be shown.

Once all the signals have been captured using this method, the designer may start building the internal behaviour of the system. This behaviour is captured expanding the use case maps start and end points into internal UCM paths. Note that this is not a static process, thus during the design phase the designer may discover errors concerning the incoming and outgoing signals. These mistakes should be corrected in an iterative way, and the affected maps redrawn. Discussions about iterative requirements engineering processes are beyond the scope of this paper.

3.1. UCM structure

In our approach, the first step to construct the UCM structure is to divide the system into subsystems. The example shown in Figure 4 illustrates the breakdown of a mobile station into components. It was broken down into the Mobile Station itself and the Network Card. For simplicity the example in this paper one will not need a further breakdown into more architectural entities. Thus it would look like Figure 8.

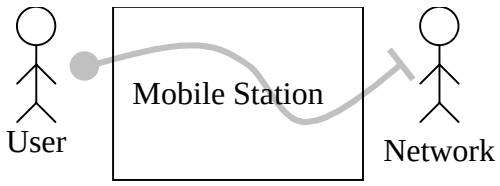

Figure 8 - Final UCM architecture

3.2. UCM behaviour

The degree to which one can determine the UCMs behaviour depends on the familiarity of the designer with respect to the UCM notation. It is not the intention of this work to guide the reader on how to design use case maps in much detail. A simple start is to identify the main steps of a use case, and describe those in terms of stubs along the sequence of actions. For instance, when a user sends the "PowerOn" signal to the mobile station, it should, 1) run its own start-up procedure, 2) "Register" onto the wireless network, and 3) start "Call Processing". In parallel to registering, the "Handoff" handling procedure should take place.

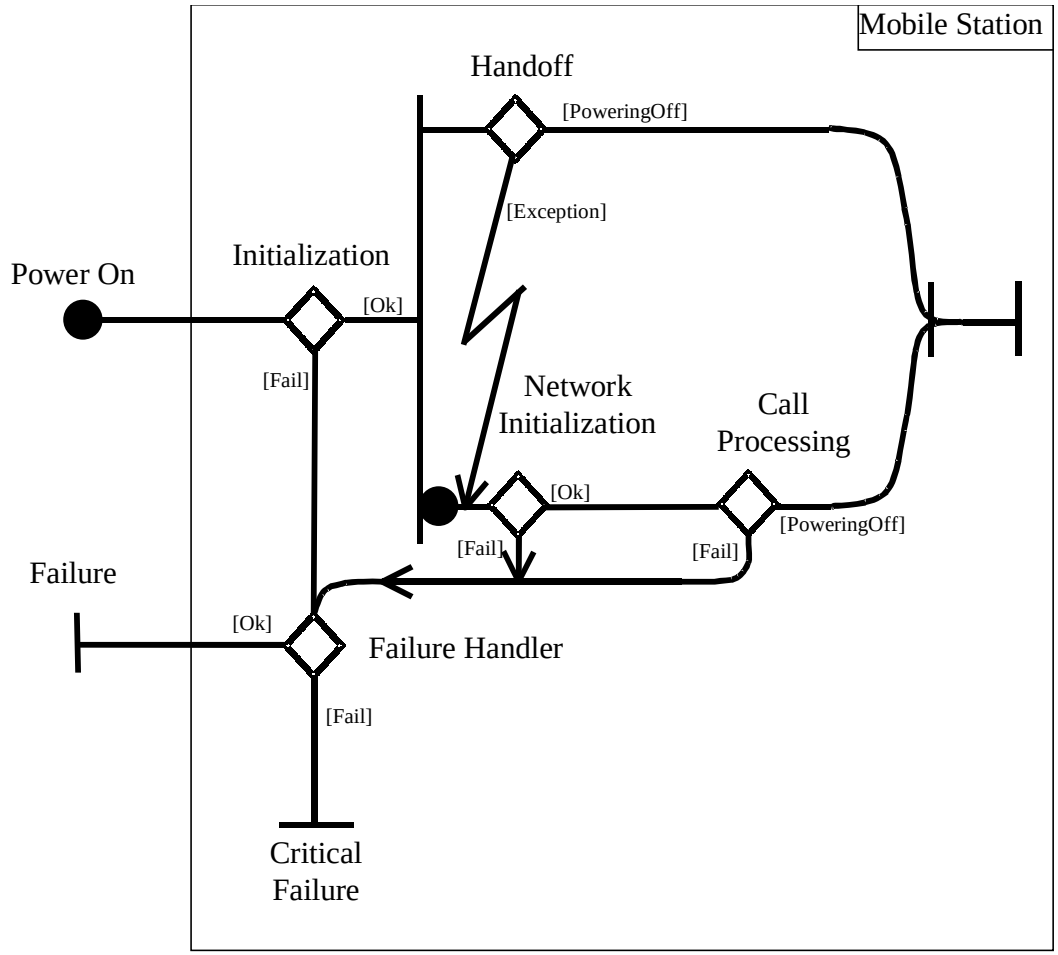

Figure 9 - High-Level Behaviour UCM model

The next step is to expand each one of the stubs and draw its proper behaviour. Note that not all observable signals need to appear in only one map. In the sub-maps there may also be more ingoing and outgoing signals. The final set of use case maps should reflect, as stated before, all the externally observable signals.

4. Describing the prototype in SDL

The derived SDL behaviour diagrams are very useful for verifying and validating the model. These two tasks are usually accomplished with a tool. Several industrial-strength tools for SDL models are available. In this section we will show the steps required to create an SDL model starting from use case maps. First we derive the architecture definition and then the

actual behaviour. Our focus is more centred on the behaviour since we are interested in functional requirements.

4.1. System Architecture

There are two activities involved in specifying a system architecture. The division of the system into blocks and processes, and the connection between these components by means of channels.

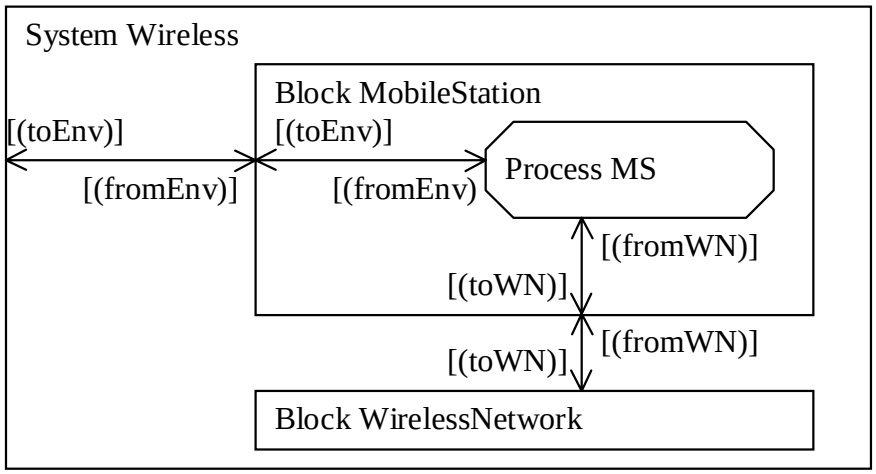

Figure 10 - The SDL overview of the architecture for a Mobile Station

Figure 10 describes the way the components will be organised inside the system "Wireless". There is a channel that communicates to the environment, i.e. the user, and it has the signal set "toEnv" and "fromEnv". The brackets indicate a set of signals differing from single signals. It has been done this way in this figure for readability and portability purposes. At the same time, the "MobileStation" block communicates with the "WirelessNetwork" block through the signal sets "toWN" and "fromWN". Channel names have been omitted for readability purposes. They are of no relevance to this work.

The channel signals are listed in Table 1 below.

ToEnv	fromEnv	toWN	FromWN
Failure, Ready, Roaming, NotRoaming, RingbackTone, NoTone, Ring, RingOff, Disconnected, NoSignal, NotPermitted, SignalError, DeviceError, PoweringOff	PowerOn, Dial, Talk, End	ConnReq, AnswReq, DiscReq	ConnConf, ConnInd, AnswInd, DiscInd, HandoffInd

Table 1 - Mobile Station signals

We can see that the architecture described in Figure 10 has more detailed than the one in Figure 8. This is due to the fact that the former one is a better way of expressing the system with the division of signals according to the receiver roles, i.e., "User" and "Wireless Network". A drawback is due to the fact that Use Cases distinguish between the external entities, while SDL and UCMs do not. A way to overcome this problem in SDL is to consider only one actor as the environment and model the others (in our case study, the network) as an internal block.

4.2. System Behaviour

For deriving the system behaviour we have found a set of SDL flowchart structures that match those of the UCMs. Responsibilities are treated as procedure calls. They may be remote procedure calls also.

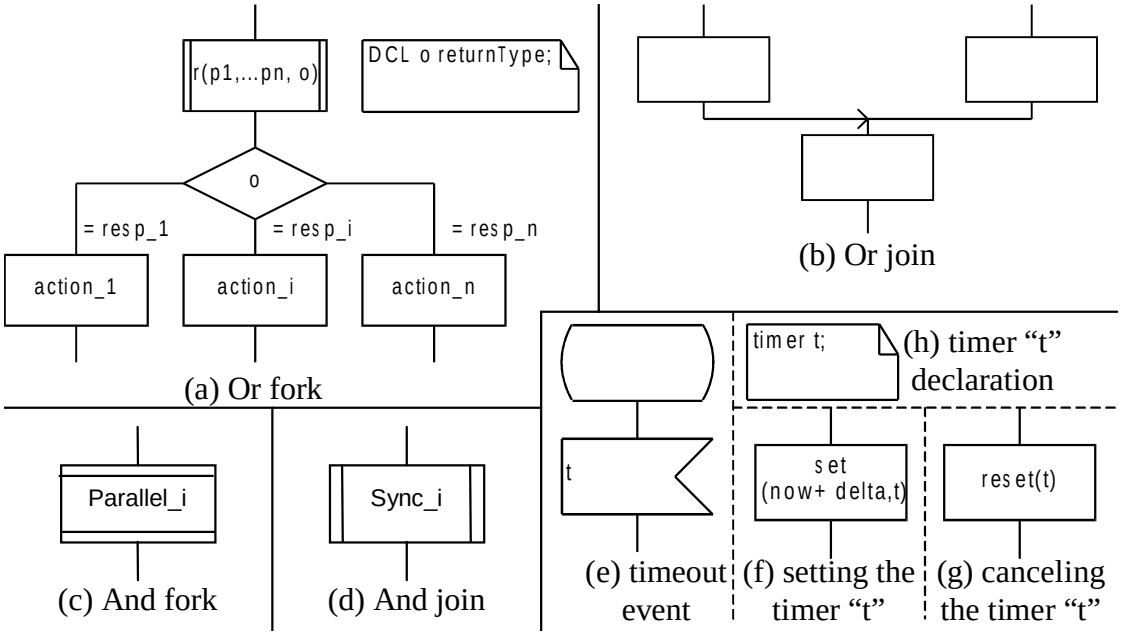

Figure 11 - SDL common structures for represeting UCMs

To represent an UCM Or-fork structure we can basically use the SDL decision symbol. In our case above, Figure 11(a), this symbol is shown with many branches. Figure 3(a) has a responsibility attached to the outgoing path, which in the corresponding SDL is represented by an "o". The way it is done is as follows: the running process, i.e., the one that represents the executing path invokes the responsibility procedure "r" with n parameters. The last parameter is the output from its execution. This output should guide the process to which branch to follow. In our case, one of the actions is the path taken.

In (b), we have the simplest case of correspondence. Any number of branches converge to one single one. In other words, whatever path has been taken will lead to the same outcome. Note that the boxes are not necessarily task symbols. They may be any valid SDL construct.

If the reader is accustomed to SDL, the timeout (e,f,g,h) will also be a very simple construct to understand. As in UCMs, to set a timer, one only needs to declare a task symbol with the "set" statement for that timer, as shown in (f). The timer must have been declared in the process, as in (h). Resetting that timer requires also a simple task with the "reset" statement as in (g). In the use case maps notation, a path is taken, usually an and-fork, to reset the timer. This entire path that resets the timer is replaced by a single task, in most cases.

To illustrate the construction of And-Forks and And-Joins we will need some more complicated constructs. The approach is to use parallel processes within the same scope. Beforehand, the designer should analyse the use case maps to ensure this procedure is needed. For example, Figure 12 illustrates a case where this construct may not be needed. The

designer may just pick one order and implement it as a single transition. Or he/she could just as well use a decision symbol with *any* (non-deterministic choice) and implement all the different combinations. For several threads this may be a problem. However, it occurs very rarely.

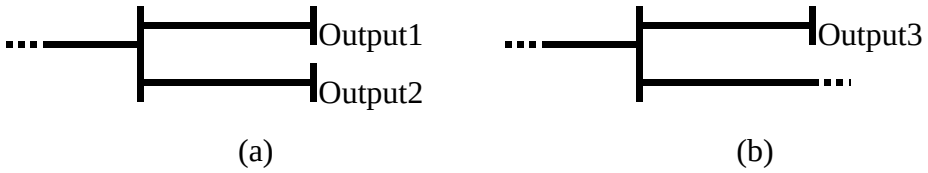

Figure 12 - Different UCM And-fork constructions

Figure 12(b) illustrates a more elaborate map. Instead of having two single endpoints, one path in the fork continues. As we have found the ordering to be scarcely necessary for two close events, again we can choose one and not even need to transform the thread into another process creation call. i.e., simply sending "Output3" and keep on executing the second path. Formally this second thread should be represented by another process. The suggestions above avoid some unnecessary work as well as improve the readability of the SDL model. This process creation construct is illustrated in Figure 11(c).

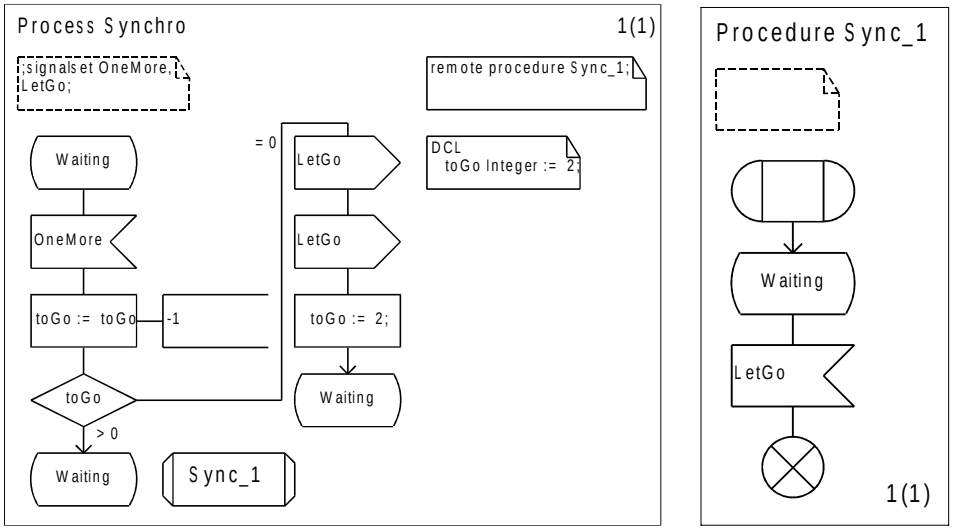

Figure 13 - The synchronization process and the Sync remote procedure

Finally the most complex construct, for our scope, is the And-Join. After experimenting a few times with different mechanisms the best way we found is to use a synchronizing process. This process' only purpose is to synchronize concurrent threads, without affecting the rest of the system. In fact, this process should not even be connected to any other blocks. Every time a thread reaches a synchronizing point, it makes a remote procedure call like the one shown in Figure 11(d). The behaviour associated with this procedure depends again on designer decisions. A typical synchronizing process is described in Figure 13. After synchronization, the processes that called the "sync" procedure should decide whether they should terminate or not. The designers should be aware that a clean way of finishing the synchronisation is to create request calls to other processes. For readability this may not be necessary, also facilitating the job. This type of construct is the mostly uncommon to find in use case maps, although it should be considered because it is a powerful UCM construct.

5. Case study: A Mobile Station (MS) basic call specification

In this section we will show the specification of a Mobile Station in use case maps and its appropriate respective counterpart in SDL. The first map has already been seen in Figure 9. It represents the root map for the others that follow. The behaviour for the SDL high level design follows. A process called “Handoff_Process” has also been created and introduced into the architecture shown in Figure 10. It is not shown due to lack of space and importance. This process communicates to the network and to the user as well as to the mobile station process. In addition, other relevant aspects of this architectural model, like the actual signal names, have been described in Table 1.

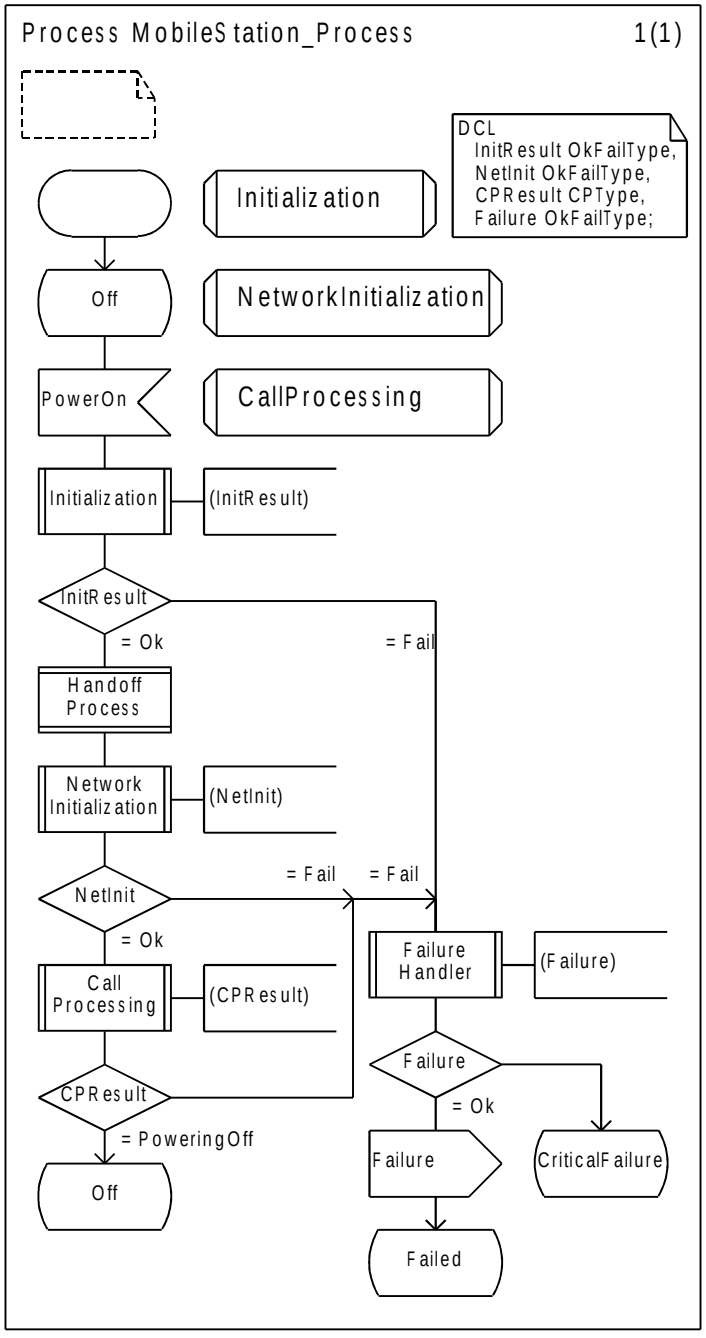

Figure 14 - The SDL behaviour for the root UCM

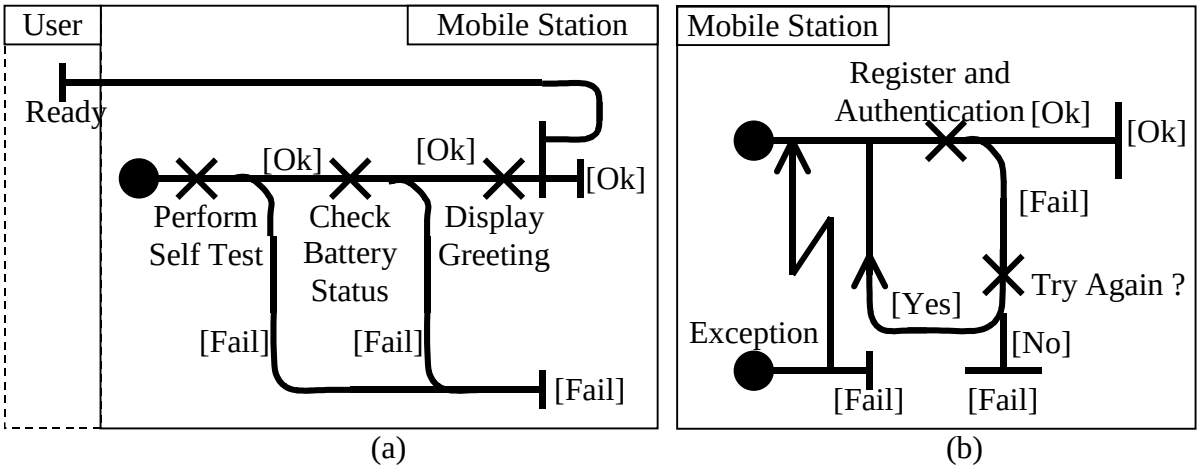

Figure 15 - The Initialization and Network Initialization UCMs

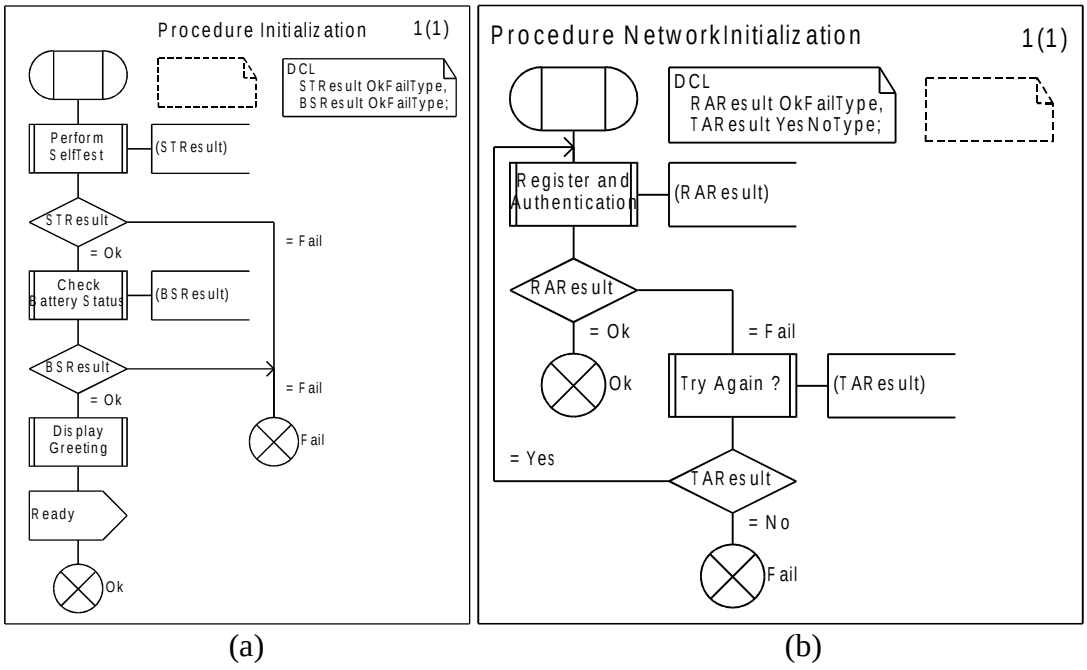

Figure 16 - The SDL behaviour for the Initialization procedures maps

Figures 14 through 20 describe in detail the complete mapping of the use case maps to SDL processes. It is straightforward to follow using the rules presented in this paper applied to figure 15 resulting in figure 16. A few design decisions were made to simplify the behavioural model. Figure 17(a) shows the failure handler plug-in. We thought it irrelevant to have separate processes to represent single signal outputs, especially because there is no concurrency in that use case map. Therefore, the solution taken was to use send symbols for the observable outputs rather than creating a parallel process for each. The final effect would have been the same. We also believe the design is clearer this way. In Figure 20 one can see that there is a decision branch after the SDL process executes the procedure “Same MSC ?”. According to the UCM in Figure 19 these two could be taken in parallel. As they only lead to two possible different situations we thought it to be easier to represent using only an *any* non-

deterministic decision symbol. Hence, we had to program it such that it had the same SDL behaviour twice, but in different orders.

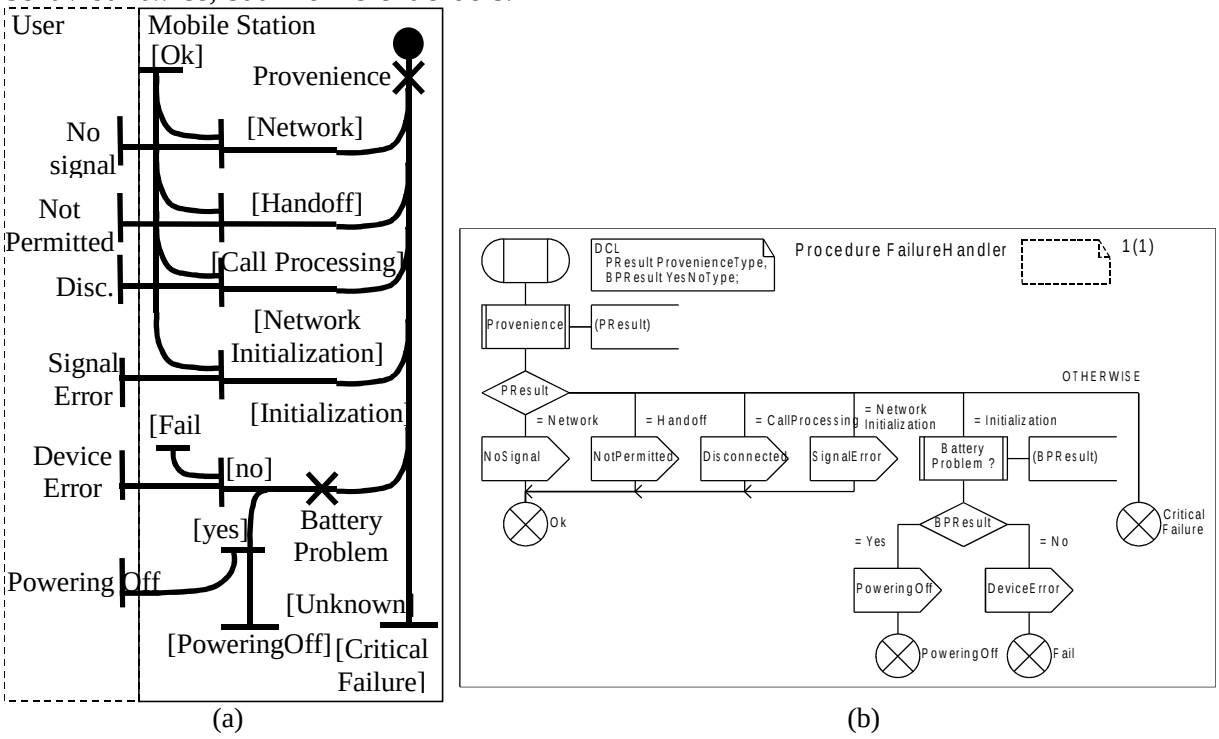

Figure 17 - The UCM and SDL behaviour for the Failure Handler

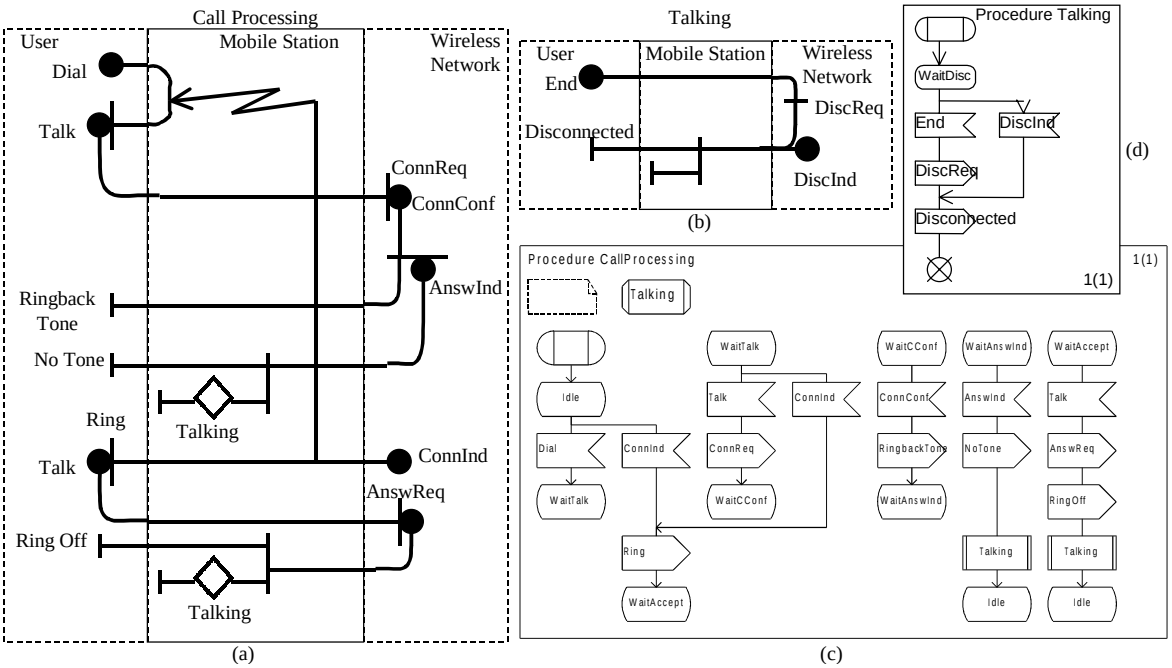

Figure 18 - The Call Processing and the corresponding SDL behaviour

Figure 18 is an example of how the behaviour for concurrent inputs and outputs would be represented in SDL. In this figure, part (c), one can see this diagram. The exception construct in part (a) is not explicitly declared in (c). The interpretation given to that exception point is of taking over the former one. Thus whenever the input “ConnInd” is received before the user

has the chance to send “Talk” a different path from making a call is taken. This is seen in SDL model as an input triggering a transition from the state “WaitTalk”.

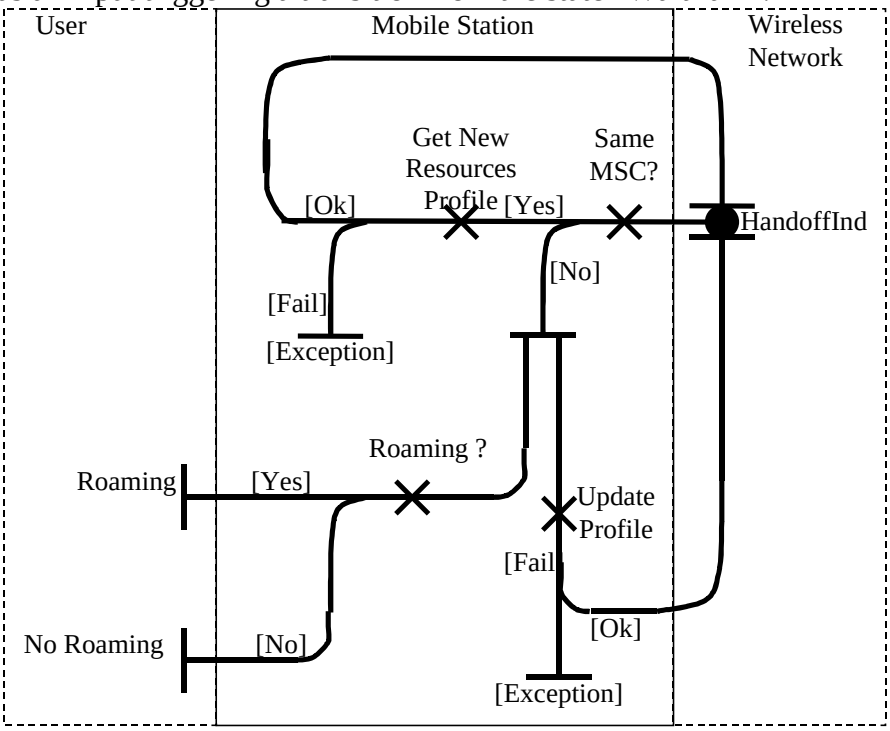

Figure 19 - Handoff use case map

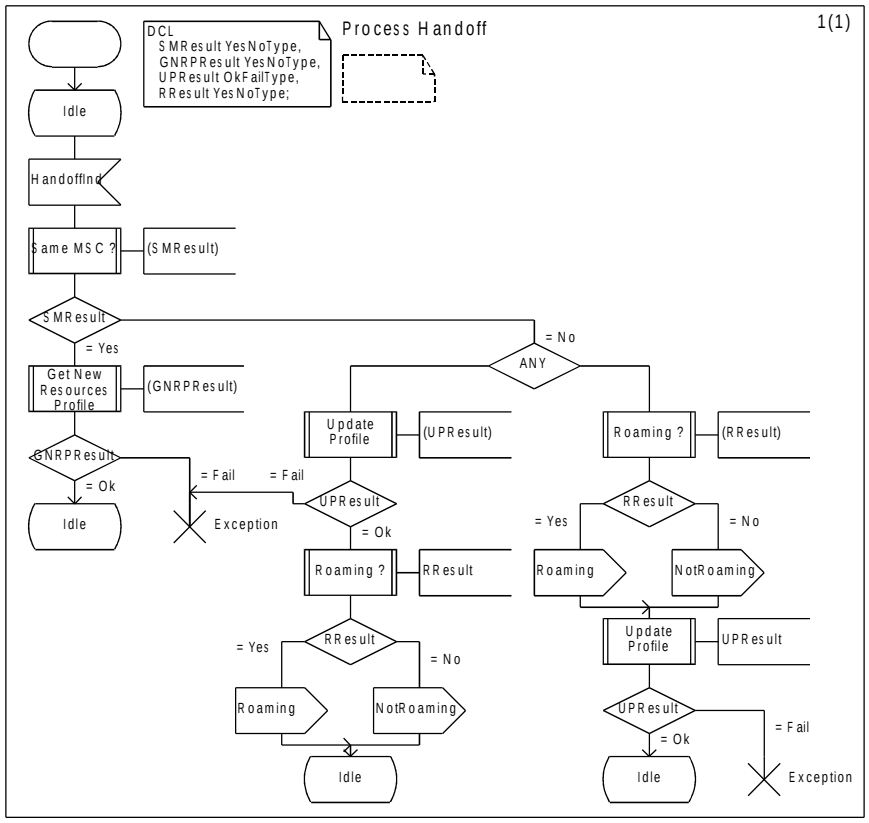

Figure 20 - Handoff SDL behaviour

6. Conclusions

We have presented a practical way of deriving high level design (in SDL) starting from a higher level behavioural and architectural model (in UCM). In our case study, we have chosen Use Case Maps because this is a fast and easy way to describe use cases at an early stage of development when having very little information about the details of the system. We have also applyed this technique to Wireless Mobile Systems development.

In summary, first we start with use cases and we draw the UCM architectural model, followed by the observable messages. Then the information in the requirements (use cases, use case diagrams) is captured in terms of general behaviour into a use case map. After reaching a level of completeness with the UCMs, the designer then will start developing the SDL model. Our methods to easily generate SDL architectural and behavioural designs were explained in detail.

Our intent is to provide a cost-effective working model such that the designer will be able to build on top of it. With the aid of CASE tools for SDL[15,16], one can promptly validate and verify it. This validation can be done by semi-automatically generating Message Sequence Charts[17] from the SDL model. As we have shown, the functional requirements of the SDL model reflect the ones from the Use Case Maps, therefore facilitating the task of ensuring correctness. Another possible applications for such a method is the automated generation of test cases[18]. Once the UCM model is ready, it is converted possibly with iterations into an SDL model. Again, with the aid of tools for generation of test suites, one can quickly generate a conformance test suite for instance.

We intend to improve this approach. These improvements are towards better techniques for capturing functional requirements with use case maps and transforming these into simulation models. We will also work on identifying the best point in time start building the bridge between the abstract model and the design model.

7. Acknowledgements

The authors grateful acknowledge helpful discussions with our colleagues in the TSERG research group at the University of Ottawa, and portrait financial support from Mitel Corporation, CITO, and the National Science and Engineering Research Council of Canada.

8. References

[1] Amyot, D., Andrade, R., Logrippo, L., Sincennes, J., Yi, Z., Formal Methods for Mobility Standards, IEEE 1999 Emerging Technology Symposium on Wireless Communication & Systems, Dallas, USA, 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/ets99.pdf

[2] Amyot, D., Logrippo, L., Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System. To appear in *Computer Communications*.

[3] Buhr, R. J. A., Casselman, R. S., Use Case Maps for Object-Oriented Systems, Prentice Hall Inc., 1996.

[4] Ellsberger, J., Hogrefe, D., Sarma, A., SDL: Formal Object-Oriented Language for Communicating Systems, Prentice Hall, 1997

[5] ITU, Z.100 (1996): Specification and Description Language (SDL), International Telecommunications Union, Geneva, 1996.

[6] Schneider, G., Winters, J. P., Applying Use Cases: A Practical Guide (Addison-Wesley Object Technology Series), Addison-Wesley, 1998

[7] Alhir, S. S., UML in a Nutshell – A Desktop Quick Reference, O'Reilly & Associates, 1998

[8] Jacobson, I., et al, Object-Oriented Software Engineering (A Use Case driven approach), ACM Press, Addison-Wesley, 1992.

[9] Selic, B., Rumbaugh, J., Using UML for Modeling complex Real-Time Systems, ObjecTime Ltd, 1998

[10] Buhr, R. J. A., Use Case Maps as Architectural Entities for Complex Systems, 1998, published on the website http://www.UseCaseMaps.org papers directory.

[11] Turner, K. J., Using Formal Description Techniques: An introduction to Estelle, LOTOS, and SDL, John Wiley & Sons, 1993.

[12] Dssouli R., Bochmann, G.V., Lahav, Y., SDL: The Next Millennium, Proceedings of the ninth SDL Forum, Montréal, Canada, Elsevier Science Ltd., Canada, 1999.

[13] Booch G., Rumbaugh, J., Jacobson, I., The Unified Modelling Language User Guide, Addison-Wesley, 1998.

[14] Buhr, R.J.A., Casselman, R.S., High-Level Design of Object-Oriented and Real-Time Systems: A Unified Approach with Use Case Maps, Prentice Hall, 1995.

[15] Telelogic AB, Telelogic TAU toolset, Malmo Sweden http://www.telelogic.com

[16] Verilog Inc., Verilog ObjectGEODE toolset, France, http://www.verilogusa.com

[17] ITU, Z.120 (1996): Message Sequence Chart (MSC), International Telecommunications Union, Geneva, 1996.

[18] Probert, R. L., Williams, A. W., Fast Functional Test Generation Using an SDL model, Proceedings of the 12th International Conference on Testing Communicating Systems, IWTCS'99, Klummer Publishers, Hungary, 1999.