The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines

F. Bordeleau and R.J.A. Buhr
Computer Science, University of Quebec, Hull, and Systems and Computer Engineering, Carleton University, Ottawa, Canada
francis_bordeleau@uqah.uquebec.ca, buhr@sce.carleton.ca

Abstract

A major problem we humans have in the engineering of complex, computer-based systems is understanding and defining how the required behaviour of a whole system is to be achieved by its components, without becoming lost in component-centric detail such as internal state machines and the intercomponent messages they trigger. This leads to other problems, such as, long iteration cycles during forward engineering while we try various detailed changes in attempts to fix erroneous system behaviour, and inadvertently introducing detailed changes during maintenance or reengineering that will damage whole system behaviour because there is a lack of backwards traceability to it from the details. This paper proposes and illustrates a design method for overcoming such problems that proceeds, in a systematic and traceable manner, from UCM models (use case maps), to MSC models (message sequence charts), to ROOM-style communicating-state-machine models. For concreteness in a limited space, the method is illustrated by a simple example that is not representative of the complex kinds of systems we have in mind.

1.0 Introduction

The fact that current development methodologies and tools only deal in a disconnected way with important properties such as abstraction, stepwise refinement and traceability results in important problems in distributed system development such as: emergence of undesired behavior, difficulty to verify the correctness of systems, undiscovered errors, long iteration to fix problems and difficulty to evaluate the impact of modifications. Also, these problems impact at different stages of system life cycle such as maintenance, extensibility and re-engineering. Ultimately, this results in important increases in development and maintenance time and cost.

In this context, we developed the UCM-ROOM design method. This method combines use case maps (UCMs) [3][4][5][6][7], message sequence charts (MSCs)[8], and ROOM-style communicating state machines [13] into a coherent method that provides both a systematic forward engineering path and backwards traceability from details to the big picture for reengineering and evolution. MSCs provide a bridge between UCM models and ROOM models over what would otherwise be a relatively large conceptual gap that results from the following differences between the models:

- UCM focuses on system behavior, while ROOM focuses on component (actor) behavior.
- UCM models abstract away from details of intercomponent communication mechanisms, while ROOM models are driven by such details.

This conceptual gap is too large to be tackled in just one step. In the UCM-ROOM methodology, the MSC modelling technique is used as an intermediate step between UCM and ROOM. The objective of the MSC modelling phase is to integrate detail-level elements in the system model on a per scenario basis before starting the definition of the component behaviors, which involves the integration of many scenarios.

The UCM-ROOM development methodology is composed of two distinct entities: UCM-ROOM modelling, which defines the set of modelling steps that are required to go from a set of scenario requirements to a detail-level ROOM model, and the UCM-ROOM development process, which mainly defines how UCM-ROOM modelling can be applied to develop large system in an iterative and incremental manner. This paper focus on UCM-ROOM modelling. Because of a lack of space, the UCM-ROOM development process will not be discussed.

The overall goal of UCM-ROOM modelling is to produce a ROOM detail-level design model of the system from requirements in a structured and systematic way. The set of modelling steps are defined in a stepwise refinement manner and they allow to keep a complete traceability between requirements and detail-level design models.

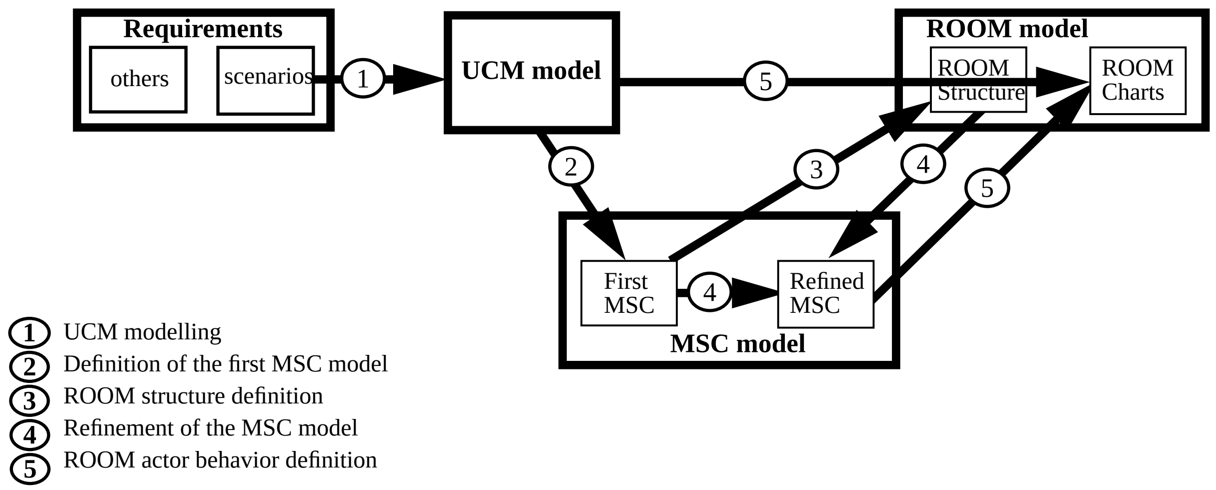
The paper is structured as follow. In section 2.0, we briefly describe the main steps that compose UCM-ROOM modelling. In section 3.0, we use a simple printer system to illustrate the application of the modelling steps. In section 4.0, we discuss how UCM-ROOM satisfies the abstraction, stepwise refinement, and traceability properties. Finally, we conclude in section 5.0.

2.0 Description of the UCM-ROOM Modelling Steps

The main UCM-ROOM modelling steps are illustrated in Figure 1. Globally, the steps are executed as follows.

1. In the UCM modelling step, a UCM high-level model of the system is produced.
2. In the first MSC definition step, the UCM model of the system is used to create a first MSC model of the system.
3. In the ROOM structure definition step, the set messages defined in the first MSC model is used to define the ROOM structure of the system.
4. In the MSC refinement step, the first MSC model and the ROOM structure model are used to create the refined MSC model.
5. Finally, in the ROOM actor behavior definition step, both the UCM model and the refined MSC model are used to define the complete behavior of each of the actors defined in the ROOM structure.

FIGURE 1. Main Steps of the UCM-ROOM Methodology


Requirements
others
scenarios
1
UCM model
5
ROOM model
ROOM Structure
ROOM Charts
2
3
4
5
First MSC
4
Refined MSC
MSC model
1 UCM modelling
2 Definition of the first MSC model
3 ROOM structure definition
4 Refinement of the MSC model
5 ROOM actor behavior definition

In this section, we briefly describe each of these steps.

2.1 UCM Modelling

The UCM phase constitutes the first phase of UCM-ROOM modelling. It makes the transition between requirements and UCM model. The objective of this phase is to produce a high-level UCM model of the system from scenario requirements. It essentially consists in capturing the different system scenarios, defined in the requirements, by means of a set of use case maps. In this phase, the UCM methodology is applied as described in [3]. The resulting set of use case maps forms the UCM model of the system. Each use case map contained in the UCM model can be directly related to a precise set of scenarios in the requirements.

Once produced, the UCM model can be used to validate the requirements.

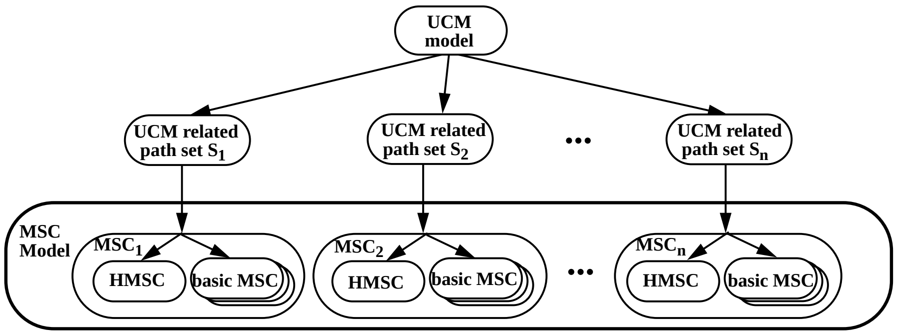
2.2 First MSC Model Definition

The objective of this step is to use the information produced in the UCM model to create a first MSC model. This step involves the description of each of the responsibilities contained in the UCM model in terms of a message sequence.

To allow to make the transition between UCM and MSC, a relationship between UCM models and MSC models is established. This relationship is illustrated in Figure 13. It associates one MSC to each related path set[1] contained in the UCM model. If a related path set is composed of more than one path segment, then the generated MSC is composed of one HMSC and a set of basic MSC (one basic MSC per path segment contained in the related path set). If the related path is composed of a single segment, then the corresponding is only composed of one basic MSC.

The establishment of this relationship between UCM and ROOM constitutes an important contribution of the UCM-ROOM methodology.

FIGURE 2. Generation of HMSC and MSC Skeleton from UCM



2.3 ROOM Structure Definition

In this step, we use the information contained in the high-level MSC model to define the ROOM structure of the system. In ROOM, a system structure is defined as a set of actors (which are visually represented using rectangles) and a set of contracts[2] that allow actors to communicate. Since at this stage the components are already defined, we only need to define the set of contracts. However, since the definition of contracts is based on protocol classes, we need to define the set of protocol classes first. Existing protocol classes can also be reused at this stage.

In order to define the global ROOM structure of the system, we, in practice, start by defining partial structures on a per MSC basis, and then merge them together in a global structure. However, for a lack of space, in this paper, we directly generate the global structure.

2.4 MSC Model Refinement

This step mainly involves two different actions: 1- add the contract identifiers, defined in the system ROOM structure definition step (section 2.3), to the first MSC model, and 2- introduce detail-level elements, such as component states, state entry and exit actions, and transition code in the MSC model. At the end of this step, a refined MSC model is produced. In this model, partial component behaviors are defined at a detailed level on a per scenario basis.

2.5 ROOM Actor Behavior Definition

The objective of this step is to define the complete behavior of each actor contained in the ROOM structure defined in section 2.3. The complete behavior model of an actor is required to satisfy all the partial component behaviors that are defined in the refined MSC model.

In the context of the UCM-ROOM methodology, this step essentially remains a creative modelling step. Some

1. We call *UCM related path set* the set of UCM paths that can be triggered from a single starting point. The concept of related path set plays a central role in the first MSC phase. A related path set is composed of a set of path segments. Related path sets constitute the basic element for the generation of MSCs from UCMs, i.e. MSCs are generated on a per related path set basis.
2. In ROOM, a contract is defined as a pair of actor ports (which are visually represented by squares placed on actor boundaries), which play the role of interface components, and a binding that links the ports. Also a port is defined as an instance of a protocol class. In this paper, for simplicity, we only consider contract that are composed of two ports that are instances of the same protocol class. However, the UCM-ROOM design methodology does not impose this constraint.

methods have been defined to automatically generate state machine component behavior models from MSC models (e.g. [9]), but we believe that they are inadequate for many reasons. The main reason is that several aspects, other than the ones described in MSC models, need to be considered in order to produce good actor behavior models. These include aspects like concurrency between scenarios, performance, extensibility, and reuse.

3.0 A Simple Printer System Case Study

In this section, we illustrate the use of the UCM-ROOM design methodology using a simple printer system. For this purpose, we first describe the requirements of the printer system, and then apply the different steps that compose the UCM-ROOM design methodology to develop the system.

3.1 Requirements

The current case study consists in developing a simple printer system which prints files on user request. The printer system itself is composed of two components: a printer driver, which is responsible for managing printer request and for feeding the printer with file characters obtained from the file system via its standard interface, and a standard printer machine. The environment of the Printer System is composed of: a user which enter printer commands, a file system by which files are accessed, and a piece of paper on which characters are printed. The file system provides the usual set of file handling functions which includes: open file for reading, close file and read next character in file.

Users may interact with the printer system by entering the two following commands:

> print *file_name*, which triggers the printing of the file file_name, and

> stop, which interrupt the printing of the file.

For the purpose of this case study, we abstract from the queueing mechanism that is required to handle multiple printing requests. We consider only one request at the time.

3.1.1 Printer System Scenarios

The current version of the printer system is concerned with the implementation of two main scenarios: a Print File scenario and a Stop Printing scenario.

Print File Scenario

The Print File scenario may be triggered at any time when the system is idle by entering the print command together with the name of the file to be printed. In order to print the file, the system first performed the initialization responsibilities which includes opening the file and requesting the printer, and then enters the printing loop. The printing loop consists in successively getting the next character from the file system and printing the character on the paper. The printing loop is exited on reception of an end-of-file message from the file system. Once end-of-file is reached, the printer is released, the file is closed, and the system is returned to its idle state.

Stop Printing Scenario

The Stop Printing scenario may be triggered at any time when the system is in the printing state by entering the stop command. On reception of the stop command, the printing loop is interrupted, the printer is released, the file is closed, and the system is returned to its idle state.

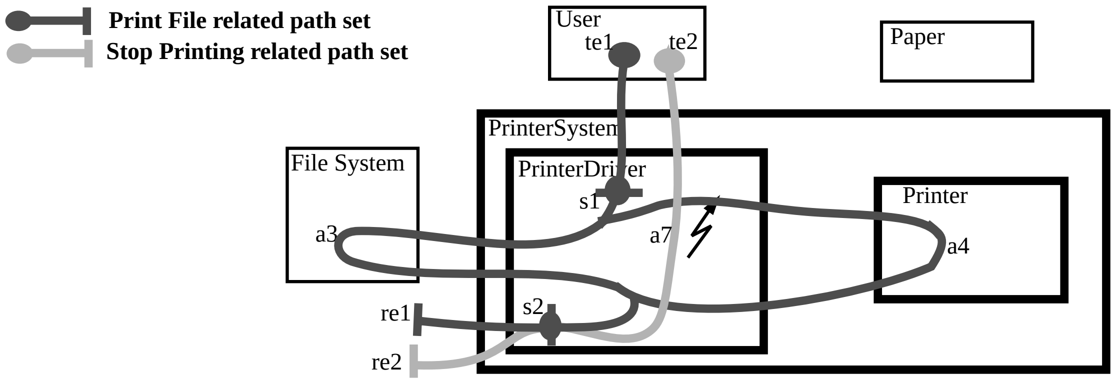
3.2 UCM Modelling

The objective of the UCM modelling step is to produce a UCM high-level model of the system. In Figure 3, the global UCM model of the printer system is given[1]. In this UCM model, the two related path sets, Print File and Stop

1. The semantics of the UCM notation is as follows. The execution of a UCM related path set starts at a starting point (●) and terminate at an end bar (▌). Alternate paths, using or-fork ( ) and or-join ( ), and parallel segments, using and-fork ( ) and and-join ( ), can also be express along paths.

Printing, that compose the printer system are composed together in a single diagram. Also, the initialization and termination responsibilities have been grouped into UCM stubs (⬮), respectively labelled S1 and S2. The UCM decomposition of these stubs is given in Figure 4. The interruption of the printing loop described in the Stop Printing scenario textual description is visually expressed using the UCM *abort* symbol (⚡). Also, since both paths contain the terminate printing stub, the two paths have been coupled at the printing stub location.

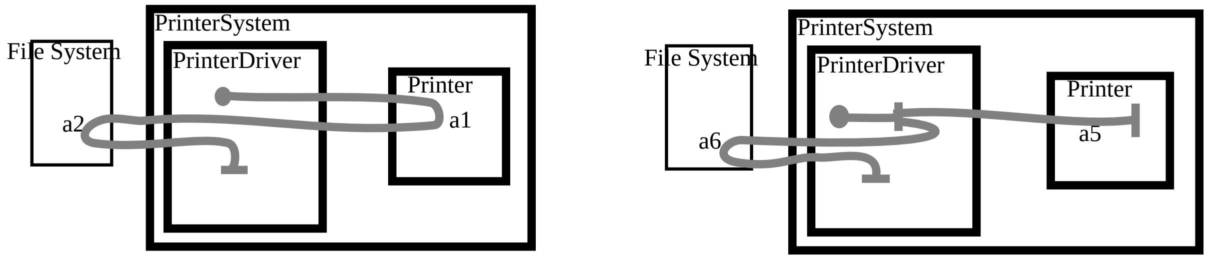
FIGURE 3. Compound UCM of the Printer System



te1: start printing triggering event, which corresponds to the print command (pre-condition: the system is in the idle state)
te2: stop printing triggering event, which corresponds to the stop command (pre-condition: the system is in the printing state)
re1: printing completed (post-condition: the system returns to the idle state)
re2: printing stopped (post-condition: the system returns to the idle state)
s1: initiate printing stub (request printer, open file)
s2: terminate printing stub (release printer, close file)
a3: get next character
a4: print character
a7: interrupt printing loop

In the UCM decomposition of stub s2 (Figure 4), the UCM and-fork notation ( ) is used to illustrate that at this level no decision has been taken yet regarding the order in which the release printer (a5) and close file (a6) responsibilities will be executed.

FIGURE 4. UCM Stubs



a1: request printer
a2: open file

UCM Stub S1

a5: release printer
a6: close file

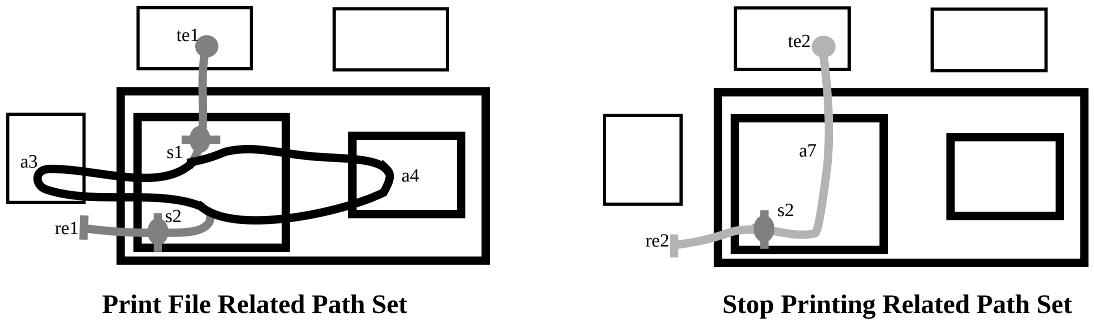
UCM Stub S2

3.3 First MSC Definition

The objective of the first MSC definition step consists in producing an MSC model of the system by describing each of the responsibility defined in the UCM model by means of a message sequence. As described in section 2.2, one MSC is produced for each related path set contained in the UCM model.

The first action required in this step consists in decomposing the UCM model into a set of related path sets. The decomposition of the printer system UCM model is illustrated in Figure 5.

FIGURE 5. Decomposition of the Printer System UCM Model

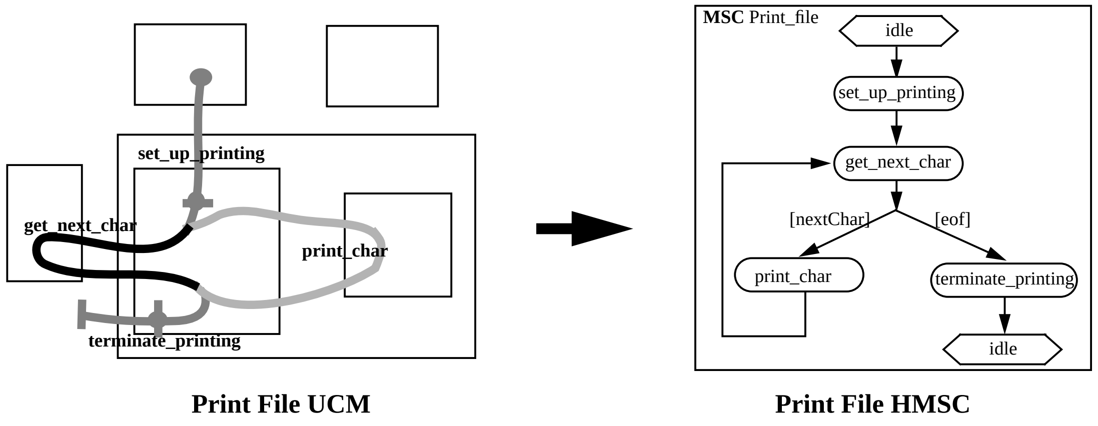


We now individually describe the definition of the first version of the Print File and Stop Printing MSCs. At this stage, in order to maintain a complete traceability between UCM responsibilities and MSC message sequences, responsibilities are placed on the component timelines in the basic MSCs (see Figure 7 for an example).

Print File MSC

Since the Print File UCM related set is composed of more than one path segments, we first need to generate a HMSC that reflects the path segment structure of the related path set. The generation of the Print File HMSC is illustrated in Figure 6. We observe, in this figure, that one MSC reference is created in the HMSC for each path segment contained in the related path set, and that the sequence of execution of the different path segments is respected in the HMSC. Also, the precondition (idle) and postcondition (idle) associated with the execution of the Print File related path set are introduced by means of initial and final conditions in the HMSC.

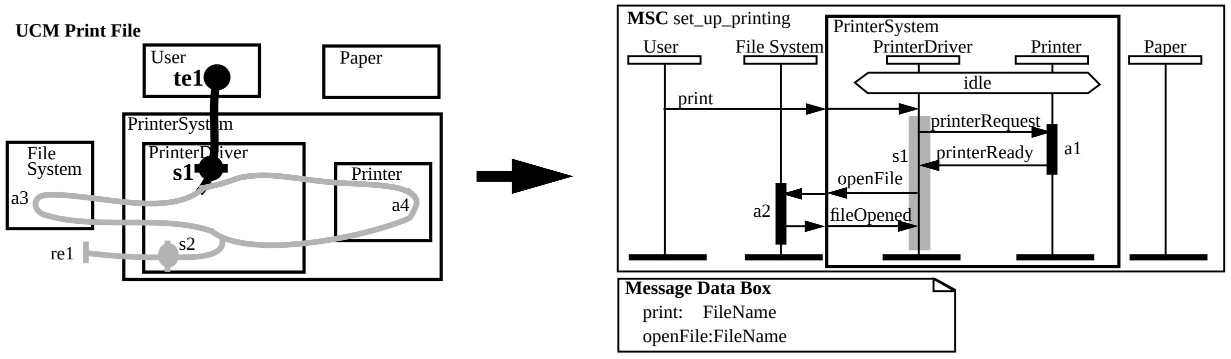
FIGURE 6. Generation of the Print File HMSC



In Figure 7, Figure 8, Figure 9 and Figure 10 the high-level basic MSCs that compose the Print File MSC are given. These basic MSCs are obtained by refining the different responsibilities expressed in the Print File UCM related path set (Figure 5) by means of message sequences. In these basic MSCs, we observe that the different responsibilities defined in the UCM model are placed on the timeline of the component to which they have been allocated in the UCM model. Also, UCM stubs and their UCM decomposition are illustrated together in detail-level basic MSCs. In such cases, the responsibilities defined in the UCM decomposition of the stub are placed within the timeline boundaries of the stub itself (an example is given in Figure 7).

In Figure 7, the set_up_printing basic MSC given. In this basic MSC, the triggering event message arrow, on which the print message is sent, is connected to Printer Driver. Responsibility a1 (request printer) is refined as two successive messages: a printerRequest message sent from Printer Driver to Printer, and a printerReady message sent from the Printer to the Printer Driver. Responsibility a2 (open file) is also refined as two successive messages: an openFile message sent from Printer Driver to File System, and a fileOpened message sent from the File System to the Printer Driver.

FIGURE 7. set_up_printing Basic MSC



In the get_next_char basic MSC given in Figure 8, responsibility a3 (get next char) is refined as a readNextChar sent from the Printer Driver to the File System, followed by either a nextChar message or a eof message sent by the File System to the Printer Driver. The alternative between the two possible messages is expressed using the MSC inline alternative expression (box with “alt” indicated in the top left corner). The inline alternative expression contains, in this case, two alternatives: one for the nextChar message, and one for the eof message.

As expressed in the HMSC of Figure 6, the basic MSC that will be executed after the get_next_char MSC depends on the message returned by the File System. If a nextChar message is returned then the print_char MSC will be executed, otherwise if an eof message is returned the terminate_printing MSC will be executed.

FIGURE 8. get_next_char Basic MSC

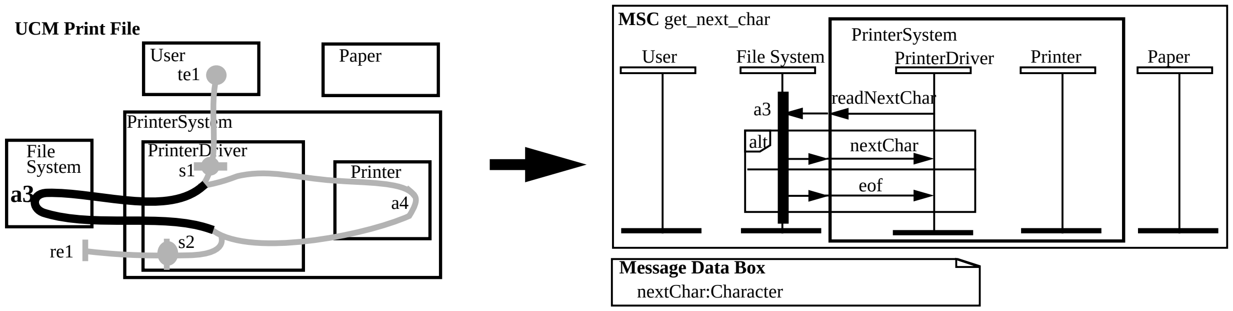


Figure 9 gives the refinement of responsibility a4 (print character). This responsibility is refined by two successive messages: a print message sent by Printer Driver to Printer followed by a char message sent from Printer to Paper.

FIGURE 9. print_char Basic MSC

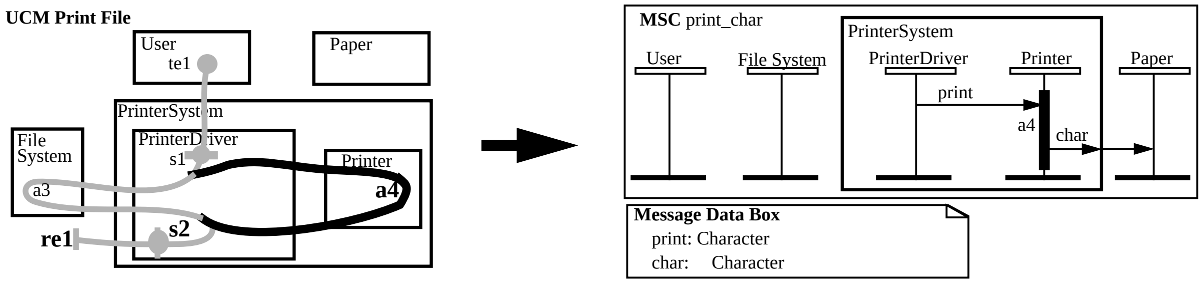
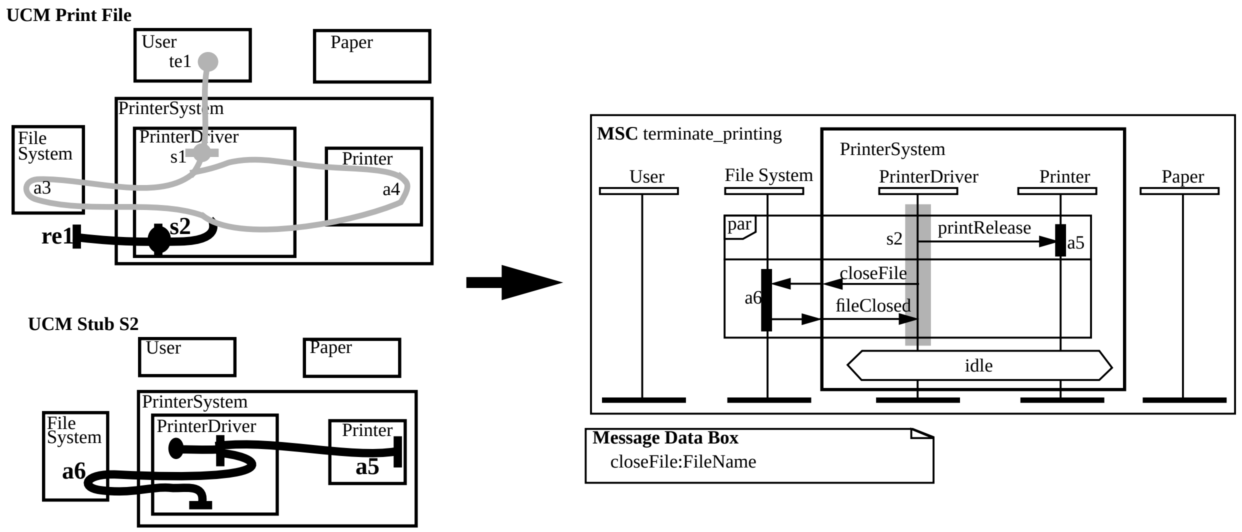


Figure 10 gives the refinement of responsibility a5 (release printer) and a6 (close file). Responsibility a5 is refine as a single printerRelease message sent from PrinterDriver to Printer, while responsibility a6 is refined by a closeFile message sent from Printer Driver to File System, followed by a fileClosed message sent back from File System to Printer Driver. We recall that these two responsibilities are executed in parallel.

FIGURE 10. terminate_printing Basic MSC

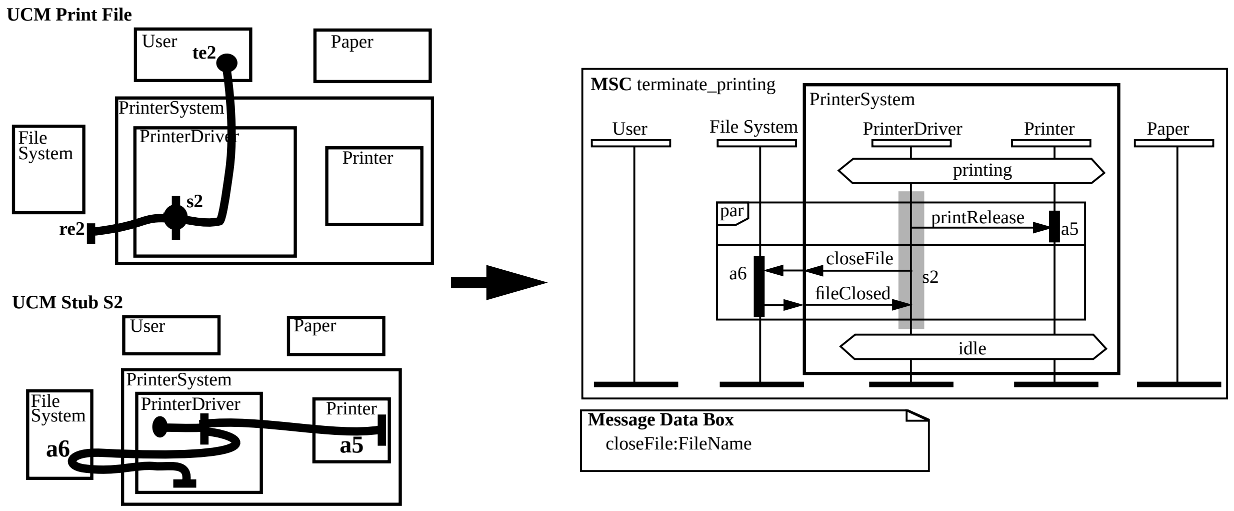


The HMSC of Figure 6 together with the set of basic MSCs described in Figure 7, Figure 8, Figure 9 and Figure 10 constitute the first version of the Print File MSC.

Stop Printing MSC

In this case, since the Stop Printing UCM related path set (Figure 5) is composed of a single path segment, we directly generate a basic MSC (the MSC is composed of a single basic MSC). The resulting Stop Printing basic MSC is given in Figure 11. This basic MSC is the same as the terminate_printing basic MSC (Figure 10) with the exception that here an initial state (printing) condition is specified.

FIGURE 11. Stop Printing basic MSC



3.4 ROOM Structure Definition

The objective of this step consists in defining the complete ROOM structure of the system. However, before we define the ROOM structure of the system, we first need to define the set of protocol classes that are required to exchange the different messages defined in the high-level MSC model produced in section 3.3. Existing protocol classes can also be reused at this stage.

Protocol Class Definition

Using the information contained in the different basic MSCs defined in the high-level MSC model (section 3.3), we define four protocol classes that will be used to define the global ROOM structure of the system. The resulting set of protocol classes is given in Figure 12.

FIGURE 12. Definition of Print File Protocol Classes

Protocol Class PrinterSystUser
```
{in:  {{print; FileName}, {stop}}
}
```

Protocol Class FileSystemInterface
```
{in:  {{openFile; FileName}, {readNextChar;
        FileName}, {closeFile; FileName}}
out:{{fileOpened}, {nextChar; Character}, {eof},
        {fileClosed}}
}
```

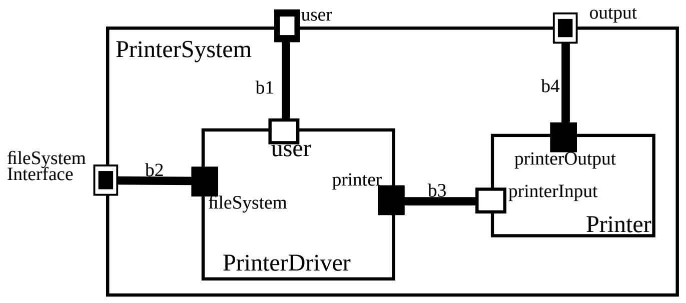
Protocol PrinterInterface
```
{in:  {{printerRequest}, {print; Character},
        {printerRelease}}
out:{{printerReady}}
}
```

Protocol Class PrinterSystOutput
```
{in:  {{char; Character}}
}
```

Structure Definition

In Figure 13, the resulting ROOM structure of the printer system is given. The contracts used in this structure are direct instances of the protocol classes given in Figure 12.

FIGURE 13. Global ROOM Structure of the Printer System



Relationship between
Contracts and Protocol Classes

Contract		Protocol Class
b1	→	PrinterSystUser
b2	→	FileSystemInterface
b3	→	PrinterInterface
b4	→	PrinterSystOutput

3.5 Refined MSC Definition

The objective of this step is to produce a refined MSC model of the system. It consists in refining the first MSC model produced in section 3.3 by adding the ROOM structure information defined in section 3.4, and by introducing detail-level elements in the MSC model.

Print File MSC

In Figure 14, Figure 15, Figure 16 and Figure 17, we give the detailed versions of the Print File basic MSCs.

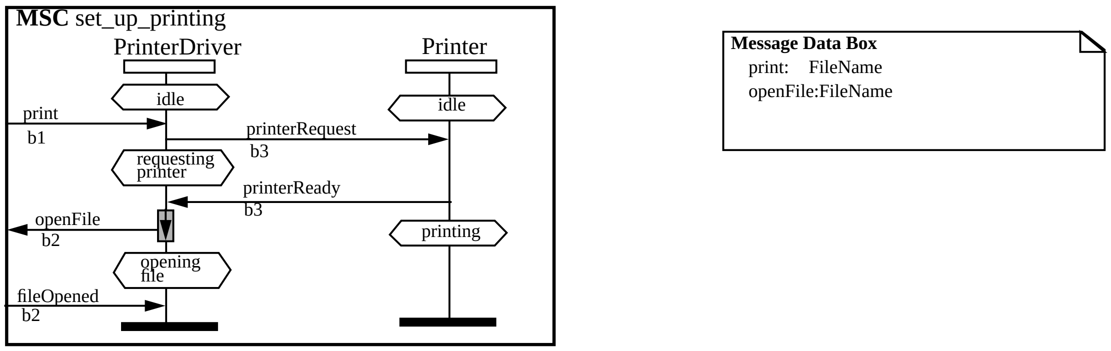
In Figure 14, the refined version of the set_up_printing basic MSC is given. In this basic MSC, two detail-level elements have been introduced: component states and state entry code.

With respect to states, we first observe that the idle global Printer System state that was defined in the previous version of the basic MSC, is decomposed into two idle components states, one in Printer Driver and one in Printer. Also, a component state is introduced in components before every incoming message arrow. For example, a requestingPrinter state is defined in Printer Driver just before the printerReady message is received.

Also, we observe that, in Printer Driver, the openFile outgoing message has been identified as an entry action (▼) in the openingFile state.

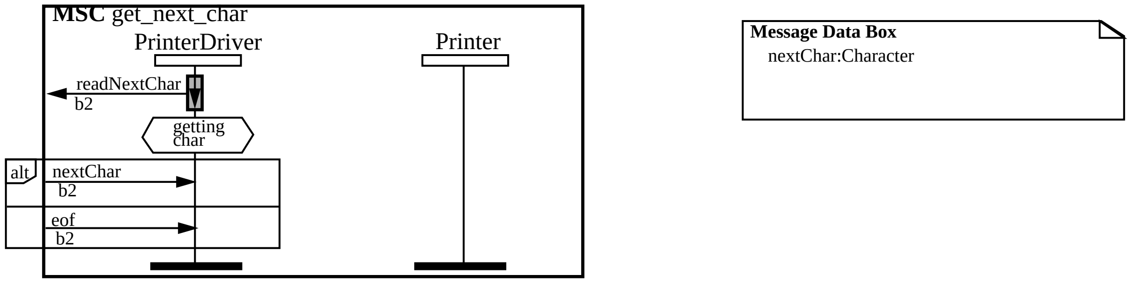
All the other outgoing messages, i.e. the ones that are not explicitly identified neither as entry action nor as exit action (▲) in the basic MSC, are considered by default to be part of the transition code. In this case, the printerRequest message constitutes such a message for Printer Driver, and the printerReady message constitutes such a message for Printer.

FIGURE 14. set_up_printing Detailed MSC



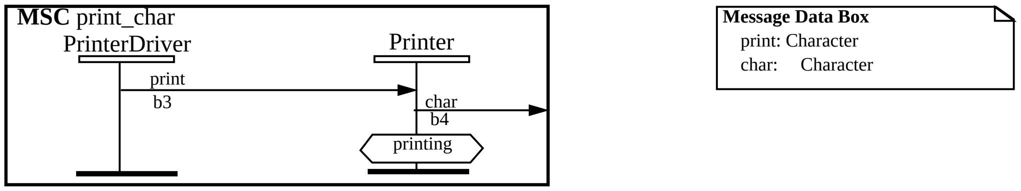
In Figure 15, the refined version of the get_next_char basic MSC is given. In this basic MSC, a gettingChar state is introduced in Printer Driver and the sending of the readNextChar message is identified as an entry action for this state.

FIGURE 15. get_next_char Detailed MSC



In Figure 16, the refined version of the print_char basic MSC is given. In this basic MSC, only a printing state is defined in Printer after the sending of the char message. This indicates that, after sending the char message, Printer goes back to the printing state previously defined in Figure 14.

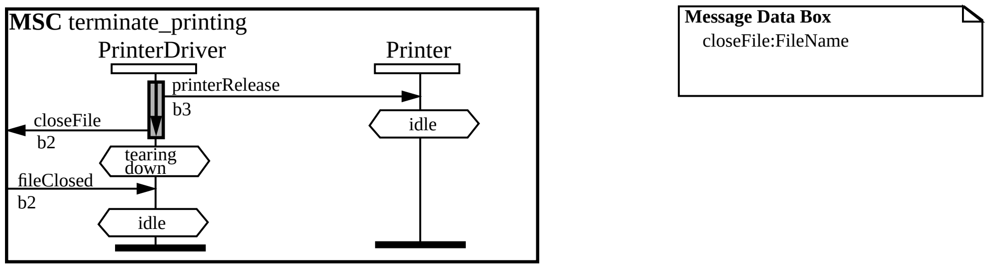
FIGURE 16. print_char Detailed MSC



In Figure 17, the refined version of the terminate_printing basic MSC is given. In this basic MSC, component states and entry-code are introduced. In Printer Driver, a tearingDown state is defined after the closeFile message is sent, and two messages, printerRelease and closeFile, are identified as an entry action in the tearingDown state. Also, it is specified that both Printer Driver and Printer return to their respective idle state after the execution of terminate_printing.

Also, we observe, in this figure, that the ordering of the printerRelease and closeFile messages that was non-deterministically defined in the previous basic MSCs using the parallel composition inline expression is now deterministic. The printerRelease message is sent first, and then the closeFile is sent.

FIGURE 17. terminate_printing Detail-level MSC

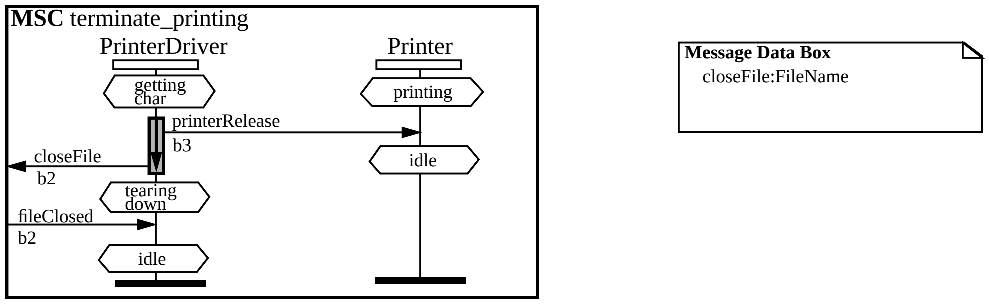


The HMSC of Figure 2 together with the set of basic MSCs described in Figure 14, Figure 15, Figure 16 and Figure 17 constitute the detailed version of the Print File MSC.

Stop Printing MSC

Figure 18 gives the Stop Printing refined MSC. This MSC is identical to the one of Figure 17 with the exception that in this one initial component state conditions are specified: Printer Driver must be in the gettingChar state and Printer must be in the printing state.

FIGURE 18. Stop Printing Detail-level MSC

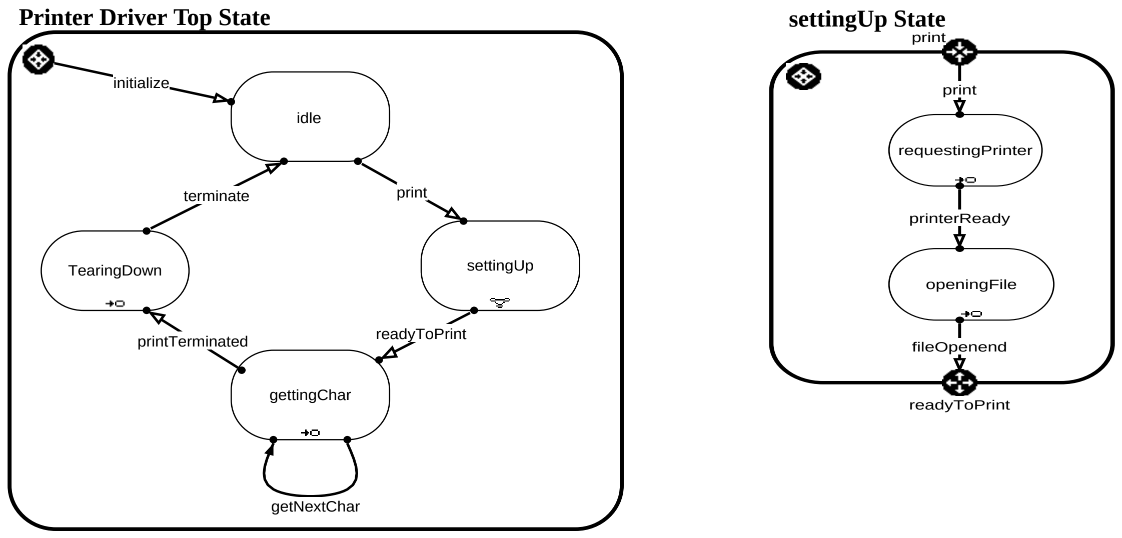


3.6 ROOM Actor Behavior Definition

In this step, the ROOMCharts behavior models of the Printer Driver and Printer Actors is defined. The actor behavior model defined in this step must satisfy all the different partial component behavior defined in the detail-level MSC model.

In Figure 19, the ROOMCharts model of the Printer Driver Actor is given. In this ROOMCharts model, the requestingPrinter and openingFile states are grouped together in the settingUp composite state.

FIGURE 19. Printer Driver Complete Behavior Model



The ROOMCharts code that corresponds to the different state transitions and state entry actions is given above. This code can be automatically generated from the different transitions and entry actions defined in the detail-level MSC model.

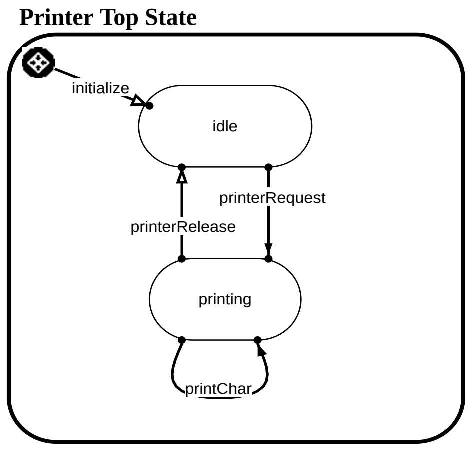
```
transitions: /*  of state top */
    {
    transition terminate/TearingDown:
        {source: state TearingDown
        destination: state idle
        triggered by:
            { event: {signals: {%fileClosed} on: {fileSystem}}}
        } /* end of transition terminate */;
    transition printTerminated/gettingChar:
        {source: state gettingChar
        destination: state TearingDown
        triggered by: { event: {signals: {%stop} on: {user}}}
        } /* end of transition printTerminated */;
    transition print/idle:
        {source: state idle
        destination: state settingUp
        triggered by: { event: {signals: {%print} on: {user}}}
            } /* end of transition print */;
    transition getNextChar/gettingChar:
        { source: state gettingChar
        destination: state gettingChar
        triggered by: { event: {signals: {%nextChar} on: {fileSystem}}}
        code: {|   | SEND printer SIGNAL %char DATA msg data ENDSEND}
        } /* end of transition getNextChar */;
    transition readyToPrint/settingUp:
        {source: state settingUp
        destination: state gettingChar
        } /* end of transition readyToPrint */;
    } /* end of transitions in: top */
```

```
transitions: /*  of state settingUp */
    {
    transition printerReady/requestingPrinter:
        {source: state requestingPrinter
        destination: state openingFile
        triggered by: { event: {signals: {%printerReady} on: {printer}}}
        } /* end of transition printerReady */;
    transition fileOpenend/openingFile:
        {source: state openingFile
        destination: state border to transition readyToPrint/settingUp
        triggered by: { event: {signals: {%fileOpened} on: {fileSystem}}}
        } /* end of transition fileOpenend */;
    transition print/settingUp:
        {source: state border from transition print/idle
        destination: state requestingPrinter
        } /* end of transition print */;
    } /* end of transitions in: settingUp */

state requestingPrinter:
    {entry action:
        {|  | SEND printer SIGNAL %printerRequest ENDSEND}
    } /* end of state requestingPrinter */;
state openingFile:
    {entry action:
        {|  | SEND fileSystem SIGNAL %openFile ENDSEND}
    } /* end of state openingFile */;
state gettingChar:
    {entry action:
        {|  | SEND fileSystem SIGNAL %nextChar ENDSEND}
    } /* end of state gettingChar */;
state TearingDown:
    { entry action:
        {|  | SEND printer SIGNAL %printerRelease ENDSEND.
              SEND fileSystem SIGNAL %closeFile ENDSEND}
    } /* end of state TearingDown */;
```

In Figure 19, the ROOMCharts behavior model of the Printer Actor is given.

FIGURE 20. Printer Complete Behavior Model



```
transitions: /* of state top */
    {
    transition printChar/printing:
        {source: state printing
        destination: state printing
        triggered by: { event: {signals: {%char} on: {printerInput}}}
        code: {|   | SEND printerOutput SIGNAL %char DATA msg data ENDSEND}
        } /* end of transition printChar */;
    transition printerRelease/printing:
        {source: state printing
        destination: state idle
        triggered by: { event: {signals: {%printerRelease}  on: {printerInput}}}
        } /* end of transition printerRelease */;
    transition printerRequest/idle:
        {source: state idle
        destination: state printing
        triggered by: { event: {signals: {%printerRequest}  on: {printerInput}}}
        code: {|   | REPLY %printerReady ENDREPLY}
        } /* end of transition printerRequest */;
    } /* end of transitions in: top */
```

4.0 Discussion

This section discusses how the UCM-ROOM modelling steps described in this paper conforms to the abstraction, stepwise refinement, and traceability properties, and also compares the approach to other methods.

Abstraction

The UCM-ROOM design methodology groups together three different design modelling techniques: UCM, MSC and ROOM. Each of these methodologies is used at a different level of abstraction in the methodology.

With respect to system and component behavior, in the UCM-ROOM design methodology the design focus slowly shifts from system behavior in the UCM phase to component behavior in the ROOM phase. In this focus shift, the MSC methodology plays a key role. It allows to make a smooth transition between UCM and ROOM, and to maintain a complete traceability between the ROOM detail-level design model and requirements.

Stepwise Refinement

The UCM-ROOM modelling steps are defined in a stepwise refinement manner. Thus, the complete set of decisions that need to be taken by developers, in order to produce ROOM models from requirements, is broken into a set of relatively small steps that require taking less decisions at each step. Each of these steps focuses on one aspect of distributed system development at the time, e.g. capture system scenarios, define the set of system components, define inter-component communication contracts, define component (actor) behavior, etc. This way, developers can focus on different issues at different steps.

In this methodology, the different models of the systems are incrementally built as descriptions are taken by developers. Thus, UCM, MSC and ROOM models of the system are successively produced. The model produced at the end of one step constitutes the starting point (input) for the next step. As a result of the whole methodology, a complete ROOM model of the system is produced.

Traceability

Traceability is the property that relates together model elements defined at level of abstraction, or in different system models. This property is essential to allow modelers to go back-and-forth between different system models in a consistent manner.

UCM-ROOM modelling steps allows to maintain a complete traceability between ROOM detail-level model and scenario requirements.

Comparison with Other Design Methods

Models at the level of message sequence charts and communicating state machines are well understand and popular in the world of system design, so need no further justification here. The new element here is   use case maps. Why do we need them? Are there not other approaches built on more familiar models that could be used instead? Here we try to answer these questions by listing some properties of UCMs and, for each, commenting on its relationship to other methods:

1. *The UCM model has the primary objective of aiding human reasoning at a high level of abstraction, as opposed to entering details into a computer tool.* Use case maps are the only diagramming technique for system design known to the authors that was shaped solely by the need for this property. Others, e.g., [11][10][13] [8][2][1][12], are shaped primarily by the need for machine-executability of design models and/or machine-translatability into code, thus forcing a commitment to details at the level of methods, functions, messages, interprocess communication, interfaces, internal state machines of components, etc., that get in the way of reasoning at a high level of abstraction. UCMs are supplementary to such detailed methods. This paper shows how UCMs may be used to supplement one of them, namely [13], with the aid of another [8].
2. *UCM models are first-class at the macroscopic level, meaning not dependent on details of components or code.* There is only one other notation that has this property, the so-called "high level message sequence charts" under development by the Z120 community [8] ([8] covers only detailed message sequence charts, but examples of proposed high level ones are given in [11]). However, this notation does not possess Property 3 and it clouds the mind's eye with boxes in the separate behaviour diagrams that look like components but are not, exacerbating the problem of mentally superimposing behaviour on structure (see Property 3).
3. *UCMs combine system behaviour and system structure into a single coherent view.* To the author's knowledge, only use case maps possess this property at a high level of abstraction. Other diagramming techniques may attempt to do it by superimposing sequence numbers on connections in structural diagrams to indicate, say, interobject or interprocess message sequences, but this requires many diagrams to present the big picture, thus clouding the mind's eye with details. Approaches that use separate diagrams, such as detailed or high level message sequence charts [8] [11][10][2] cloud the big picture by forcing humans to combine diagrams in the mind's eye. (The term "message sequence chart" is being used here in a generic sense to cover any diagram in which parallel lines representing abstract timelines for components are connected by sequences of arrows representing intercomponent communication scenarios.)
4. *UCMs express "morphing" compactly, without requiring sequences of snapshot diagrams of changing structural forms.* Morphing is the changing stuctural form of a system as its components and intercomponent relationships change over time. To the authors' knowledge, only use case maps possess the property of representing morphing without sequences of snapshots. They do it by enlarging the concept of a component to include slots that identify places where actual components may appear (to play local roles) or disappear, by using use case paths as the loci for movement of actual components to or from slots, and by staying above the level of changing intercomponent communication paths.
5. *UCM diagrams are easily grasped as visual patterns for a system as a whole.* Use case maps can combine many behaviour patterns in single diagram in a way that enables the mind's eye to sort them out. Recognizing behaviour patterns in superimposed sequence numbers or separate detailed message sequence charts is much more difficult, particularly because many diagrams must be viewed.
6. *UCMs provide macroscopic system views for forward engineering, reverse engineering, maintenance, evolution, and reengineering.* Only use case maps and high level message sequence charts provide reference views that are independent of details and so can be used to guide decisions about details. Use case maps do it more compactly and simply (Properties 2-5).
7. *UCMs can be saved for documentation and maintained without unreasonable effort.* The avoidance of commitment to details and the compactness of use case maps contributes to this property. However, tool support is desirable (a use case map editor is currently being developed for this purpose).

It is important to remember that use case maps do not replace the other techniques referred to above, but supplement them to give a higher level view.

Although we have not used the term *architecture* up till now [14], we believe that the properties described above make use case maps a new, useful and practical form of architectural description [6].

Although this paper has presented a forward engineering example, we hope that our readers will find it as obvious as we do that any techniques that helps provide stepwise development, abstraction, and traceability in forward engineering will also help in the bacwards direction with maintenance, evolution and reengineering ([4][5] provide examples of the application of UCMs in the backwards direction).

5.0 Conclusion

In this paper, we presented the UCM-ROOM design method as a solution to some major problems we humans have in the engineering of complex, computer-based systems. We humans have difficulty understanding and defining how the required behaviour of a whole system is to be achieved by its components, without becoming lost in component-centric detail such as internal state machines and the intercomponent messages they trigger. This leads to other problems, such as, long iteration cycles during forward engineering while we try various detailed changes in attempts to fix erroneous system behaviour, and inadvertently introducing detailed changes during maintenance or reengineering that will damage whole system behaviour because there is a lack of backwards traceability to it from the details. The UCM-ROOM method provides a set of steps that provide a systematic and traceable progression from abstract UCM models (use case maps), to MSC models (message sequence charts), to ROOM-style communicating-state-machine models. Thus the method smoothly integrates three important properties, abstraction, stepwise refinement, and traceability. For concreteness in a limited space, the method was illustrated by a simple example that is not representative of the complex kinds of systems we have in mind.

6.0 Acknowledgments

NSERC, TRIO, Nortel Technology, and ObjecTime provided financial assistance. Bran Selic, Carmine Ciancibello, and Luigi Logrippo provided many helpful comments. Fahim Sheik, Julian Kung, and students from Carleton's 94.586 course tried out some of these ideas and provided feedback.

7.0 References

[1] G. Booch, Object-Oriented Design, Benjamin/Cummings, 1994.

[2] Grady Booch, James Rumbaugh, *Unified Method for Object-Oriented Development*, Documentation Set, Version 0.8, Rational Software Corporation, 1995.

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

[4] R.J.A. Buhr, A. Hubbard, *Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE-Framework Application*, Hawaii International Conference onSystem Sciences, Jan 7-10, 1997, Wailea, Hawaii, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss-final-public.ps

[5] R.J.A. Buhr, R.S. Casselman, T.W. Pearce, *Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework*, SCE 95-17, http://ftp.sce.carleton.ca/UseCaseMaps/dpwucm.ps.

[6] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/attributing.ps.

[7] R.J.A. Buhr, *Design Patterns at Different Scales*, presented at PLoP96, Allerton Park Illinois, Sep 96. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps.

[8] ITU (1995), Z.120 Message Sequence Chart (Draft Recommendation).

[9] K. Koskimies et. al., *On the Role of Scenarios in Object-Oriented Software Design*, Technical Report A-1996-1, Dept. of Computer Science, University of Tampere, Tampere, Finland.

[10] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.

[11] B. Regnell, M. Andersson, J. Bergstrand, *A Hierarchical Use Case Model with Graphical Representation*, Proc. ECBS96, IEEE Second International Symposium and Workshop on Engineering of Computer Based Systems, March 1996.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modelling and Design*, Prentice Hall 1991.

[13] B. Selic, G. Gullickson and P.T. Ward, *Real-time Object-Oriented Modeling, Wiley*, 1994.

[14] Shaw and Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.