INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

A Use Case Driven Validation Framework and Case Study

by

Anandra Boni Bangari

A thesis submitted to
the School of Graduate Studies and Research
in partial fulfillment of the degree of

Master in Computer Science

School of Information Technology and Engineering
University of Ottawa
(Ottawa - Carleton Institute for Computer Science)

ACKNOWLEDGEMENTS

I would like to express my appreciation and gratitude to my thesis supervisor Professor Robert L. Probert for his support, encouragement, and financial help throughout my thesis research. I am also grateful towards Bran Selic, Vice President Research at ObjecTime Ltd., for his help and fruitful ideas and comments. I would also like to thank Alan Williams and Daniel Amyot for their unconditional help and useful suggestions.

My deepest love and respect go to my family, whose support and understanding have helped me pursue my education in Canada.

Finally, I would like to thank the Telecommunications Research Institute of Ontario (TRIO) and ObjecTime Ltd. for providing me with the opportunity to complete an Industrially Specified Research Program (ISRP), which engendered this thesis. I also gratefully acknowledge their financial support.

To my family

ABSTRACT

In this thesis we propose a use case driven strategy for validating software systems which are built using an incremental development process. We use the Real-time Object Oriented Modeling (ROOM) development process as an example. Our technique supports validation activities whereby a partially built system is executed and tested against some desired behaviour derived from the requirements, before further refinement is made. To represent the desired behaviour, we propose a new notation called Use Case Trees (UCT), which is derived from the Tree and Tabular Combined Notation (TTCN). In UCT several scenarios can be compacted into a single use case. Each scenario can be represented by a Message Sequence Chart (MSC). We built a tool to automatically generate selected MSCs from a UCT. The generated MSCs are used to drive the system, thus checking whether its behaviour conforms to the requirements. This validation strategy is cost-efficient as it helps to catch errors at an early stage of system development.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

In the year 1997, we are able to clone mammals, ready to send people to Mars, but still we have not found a way of producing software which is free of errors. There are no techniques yet for producing an executable system out of some simple descriptions of what we want the system to do. To get from these descriptions to the executable system, we still have to go through many complicated stages of development. And of course, since there are human beings involved, this process is bound to produce inconsistencies, typing errors, bad judgement and many more types of flaws. Our ultimate goal is to satisfy the customers by providing them with a sound and reliable software system. It is not desirable to get complaints saying that the software is not working properly or it is not doing what it was supposed to do. To avoid these situations, a software system has to be thoroughly tested.

Even after having tested a system thoroughly, we can never say that it is flawless. This is because it is impossible to do exhaustive testing, in which all possibilities are considered. The goal is to optimize testing by minimizing the number of tests and maximizing coverage. Testing usually represents a fairly big percentage of what is invested in a software project; this is understandable since we all know that software which contain a lot of errors can be very costly in the end. Flight systems, for example, have to undergo a lot of testing since there are human lives at risk.

Because testing is so important, we decided to make it the subject of this thesis. Our work is focused on one broader activity of testing known as validation, which is described in chapter 2.

1.2 Objectives and Contributions of this Thesis

Our main objective is to develop a means of representing and structuring use cases in a way that is useful for the validation of software systems which are built using an incremental development process.

In this thesis we will first propose a new notation for writing desirable sequences of actions in a system, commonly called use cases, which we named Use Case Trees (UCTs). The notation is formal and simple to write and understand, qualities which differentiate it from existing notations. Right now there is no standard for writing use cases. We hope that our notation will contribute to unifying the existing notations, and help in the standardization of use cases.

We will then see how our notation can be used for validating systems which are built using an incremental development process. For this purpose, we had to convert our notation to Message Sequence Charts (MSCs) which is another notation for representing use cases, and for which we developed a tool. All this is part of the validation framework that we are proposing, and which will target incremental iterative validation of a system with the requirements, captured in our new UCT notation.

We hope that this work will add to what has already been done and at the same time provide a basis for future work.

1.3 Organization of this Thesis

In chapter 2 we will describe our current work in more detail, explaining the important terms and justifying our motivation and approach. In chapter 3 we will describe ROOM, a software development methodology that we are going to use later in our case study. In chapter 4 we will describe our new notation, followed by the use case driven validation framework that we are proposing, in chapter 5. In chapter 6 we will describe our tool which converts UCTs to MSCs. Chapter 7 will describe a case study done to assess our incremental iterative validation strategy. Chapter 8 will conclude our work, describing some future work that can be done. A glossary is added to help the reader with some terms and acronyms. Finally, Appendix A will describe the grammar for our notation, and Appendix B will show the source code for our tool.

Chapter 2

Overview of Current Work

This chapter gives an overview of our current work, along with some main concepts of software development and some related work. We found it helpful for readers who are not too familiar with some software engineering terms such as software development, testing, quality assurance, use cases, and so on, to explain them here. This will facilitate the understanding of the subsequent chapters. In this chapter the main terms of the title of this thesis, namely "use case driven" and "validation framework" are also explained. At the end of this chapter a simple fax system is described. We found that the use of a single example throughout the thesis would be very helpful to the reader.

2.1 The Software Life Cycle

To build good and reliable software there are certain steps that have to be followed. These steps can be seen as the "life cycle" of the system to be built. The first structured and widely accepted life cycle is known as the "waterfall model" and was defined as early as 1970 by Royce [Royce70] and later refined by Boehm [Boehm76] in 1976 to help cope with the growing complexity of the software projects being tackled. The waterfall model demands a systematic, sequential approach to software development that begins at the requirements level and then progresses through design, coding, testing and maintenance. Each phase should be finished before the next one can be started. However, some variations of the model such as "waterfall model with limited feedback" and "waterfall model with unlimited feedback" allow a return to a previous phase for possible modifications. Figure 2.1 shows the waterfall model with limited feedback.

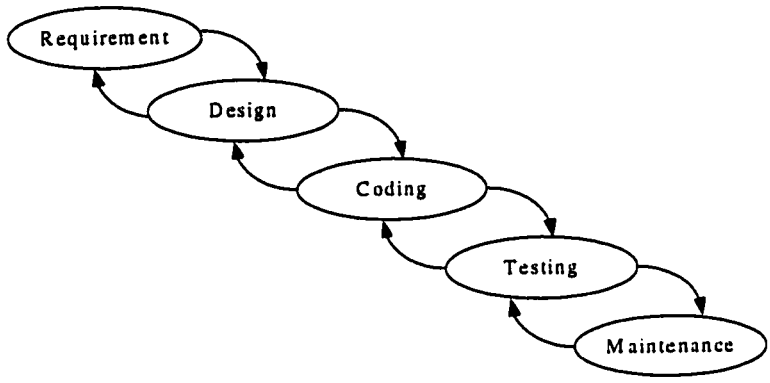


Figure 2.1

A requirement describes part of a system, either textually or graphically, giving information about its structure and behaviour. Usually all the requirements are gathered in a document before moving to the next phase, the design. Software design is basically the conversion of the requirements into a form that is more concrete, which can be assessed for quality before coding begins. Coding is the translation of the design into a language that is understood by the computer. Then comes the testing phase.

The testing phase is one of the hardest. In the three previous phases, there could have been all kinds of mistakes, such as badly captured requirements, wrong design or mistakes in the coding. Very often, errors are carried forward from phase to phase. The purpose of testing is to find as many errors as possible and remove them correctly. There are many different kinds of software testing techniques and several books have been written to discuss them [Myers79, Beizer90].

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). *Verification* refers to the set of activities that ensure that the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. *Validation* refers to a different set of activities that ensure that the software built satisfies the requirements. Boehm [Boehm84] and later Jacobson [Jacob92] state this in another way :

Verification : "Are we building the system right ?"
Validation: "Are we building the right system ?"

The basic objectives of verification and validation are to identify and resolve software problems and high-risk issues early in the software life cycle. The definition of verification and validation (V&V) encompasses many of the activities we refer to as software quality assurance (SQA). Quality assurance is an essential activity for any business that produces products to be used by others. There are several factors that ensure quality, two of the most common being *correctness* and *completeness*. These two terms are better explained if the negation is considered. A software system is *incorrect* if at least one of its behaviours does not satisfy the requirements. A software system is *incomplete* if at least one of the requirements is not satisfied by its overall behaviour.

Let us consider the simple example given below. Suppose for a given system there are ($R_1, R_2, ...,R_m$) requirements and ($B_1, B_2, ...,B_n$) possible behaviours, where $m>0$, $n>0$

The system is correct if $\forall B_i \ \exists R_j$ such that B_i satisfies R_j , where $1 \leq i \leq n$, $1 \leq j \leq m$

The system is complete if $\forall R_i \ \exists B_j$ such that B_j satisfies R_i , where $1 \leq i \leq m$, $1 \leq j \leq n$

If for a given system there is a finite set of requirements, completeness checking can be done. Correctness checking, on the other hand, is harder, if not impossible, because of the large number of possible behaviours of the system. Note that here completeness and correctness refer to the system under construction and not to the requirements. In our work we focus only on completeness checking.

As said earlier, the waterfall model has some disadvantages, and consequently is nowadays decreasing in popularity. Since testing is so important, it is not recommended to wait for all the coding to be done before starting it. It is more economical, in both time and money, to find flaws as early as possible. A number of alternatives have been considered such as the "spiral model" [Boehm88], the "rapid prototyping technique" [Goma90] and the "fourth generation technique (4GT)" [Cobb85], which are incremental development processes. They put a lot of emphasis on continuous verification and validation.

All the software development models stated above have a well-defined structure, and verification and validation form an integral part of the whole process. Since verification and validation are activities that can be planned in advance and conducted in a systematic way, a template, or a framework, can be defined which will do all these activities. However, since we are considering only validation activities in our current work, we will refer to this framework as a validation framework. In the next section the main elements of a validation framework are described.

2.2 A Validation Framework

A validation framework defines a strategy for the validation process and all the process components. It describes all the steps involved, such as the validation methods, the validation process flow, and the production and selection of tests. An example of a basic validation framework is shown in figure 2.2.

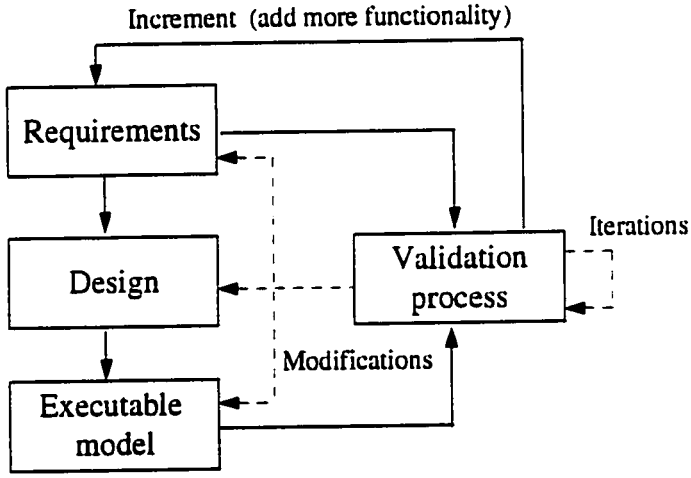


Figure 2.2

The solid lines show the normal process flow and the dotted lines show possible modifications after validation. The validation framework usually includes the development cycle, so that the different phases can be linked together. For example, validation cannot be done before a model of the system is obtained. Validation activities require the model

to be executed, and the results obtained, to be compared with the requirements. The earlier we can get an executable model of the system, the more effective will be the validation, because failure-causing errors will not be carried forward. Since we are considering incremental development processes where software systems are built step by step, with added functionalities on the next cycle, we have to do incremental validation. However, a model may not correspond to the requirements after validation. If this is the case the proper modifications have to be made and the validation repeated again. We refer to this process as iterative validation.

One of the main parts of testing is to define tests and know how to manage them. A test is an input to the system which will give a certain output. Usually the expected output is known in advance and it is compared to the actual output. If they match, the test is a pass, otherwise it is a failure. The input and the expected output, together, is called a *test case* [Myers79]. Tests can be meant for either verification or validation. Verification tests are usually targeted towards the code whereas validation tests deals more with the behaviour of the model.

In a validation framework, we should be able to provide inputs for the model of the system which is being developed, and compare the output with the expected output. We should also be able to choose which inputs will be used for validation. The inputs are usually derived from the requirements.

In chapter 5 a validation framework which incorporates an incremental iterative validation strategy is presented. At the end of the chapter an assessment is made of the validation strategy.

2.3 The Use Case Driven Approach

In the previous section we talked about the tests, in the form of inputs, that must be designed and written, in order to verify or validate a system. As explained above, verification tests are different from validation tests, and, for the purpose of this thesis, we will consider only validation tests. They are used to check whether the system under construction satisfies the requirements. At present there are no standards for writing these tests.

In 1992, Ivar Jacobson introduced the *use case* concept [Jacob92]. He defines a use case as being a complete course of events initiated by an actor, specifying the interaction that takes place between the actor and a system. An actor here is the user of the system, and hence the term use case. The purpose of a use case, according to Jacobson, is to describe the functionality of a system, before it is actually built. This is done by describing a set of use cases for each actor, by using a normal descriptive language, English in our case. A use case can describe the normal behaviour of a system, called the *basic course*, but also, variants of the basic course, most of the time exceptional behaviour, called the *alternative courses*. Normally a use case has only one basic course, but several alternative courses.

The use cases defined by Jacobson show only a high level view of a system. Only the actors (users) and the system are considered, with no concern about the internal behaviour of the different entities which make up the system. However, this point of view may not be too useful when designing the internal entities of the system. For this reason, we will view a use case as being a sequence of interactions between any entities that make up a system. If two entities are interacting, one is the "user" of the other, and vice versa. In this way we can describe a system in more detail.

A use case can have a basic and several alternative courses, and we will consider each course as being a scenario. So a *scenario* is a single sequence of interactions between any entities of a system. We view a use case as consisting of one or more scenarios.

The primary purpose of a use case was to describe a system before it was built. But we believe that use cases can be used in another way. If the system is partially or totally built, we may use use cases to describe its expected behaviour. Our goal will then be to check whether the system behaves as the use cases. We refer to this approach as the "use case driven approach". This is a validation-directed development process whereby use cases are used for testing. Use cases are derived from the requirements and are written using a particular notation. They are then used to derive inputs which are going to drive the system or a model of the system. The system should behave as described by the

use cases or otherwise there is a flaw in the system. Note that in this approach the use cases are not used to build the system, but rather, to validate it.

In chapter 4 we will present a notation for writing use cases called Use Case Trees (UCT). This notation is derived from the Tree and Tabular Combined Notation (TTCN) [TTCN92], a test case description language. In chapter 6 we will show how Message Sequence Charts [MSC92], the inputs to drive the system, can be derived from the UCT notation.

2.4 The use of ObjecTime for our Case Study

ObjecTime is a toolset which helps to develop systems using a modeling language known as Real-time Object Oriented Modeling (ROOM) [Selic94]. The development of a system using ObjecTime is an incremental process and hence there is continuous testing which can be done, and furthermore, code is automatically generated from the design.

ObjecTime is a fairly new toolset and for this reason, the testing framework is still under development. Verification activities are of lesser concern than validation activities because, since code is automatically generated, the chance of errors in the code is lessened. What is more important is to check whether the system which is constructed using the tool, behaves as expected, by checking its behaviour with the requirements. Right now this is done manually and obviously, is very costly.

For these reasons, ObjecTime became a perfect choice for our case study. To build a validation framework we had to choose a particular software development life cycle, which in our case is the incremental development process of ObjecTime. Furthermore, to use the use case driven approach we needed to focus only on validation activities, and this is required in the ObjecTime toolset.

2.5 Related Work

During the past few years, a lot of research has been done in areas which are related to our current work. It is commonly accepted that use cases are of prime importance for the description of the behaviour of a system but no one has yet come up with a notation which would be a good candidate for standardization.

In 1996, Booch [Booch91], Rumbaugh [Rumb91] and Jacobson [Jacob92] unified their work to produce the Unified Modeling Language (UML) [UML97]. The unified modeling language is a language for specifying, constructing, visualizing and documenting the artifacts of a software system. In this language, use case diagrams and prose descriptions, which were proposed by Jacobson [Jacob92], are used to describe use cases. Use case diagrams show message interactions between the system and its users, only. The diagram does not show anything about the internal structure or behaviour of the system. The prose descriptions, on the other hand, tend to be too lengthy and ambiguous.

Another notation which is used to describe message interactions is Message Sequence Charts (MSC) [MSC92, Ander95, Regnell96]. An example of a MSC is given in chapter 3. A MSC describes a single set of message interactions between the entities of a system. Because of this restriction, High-level Message Sequence Charts (HMSCs) [MSC96] have been developed, which can describe more than one set of message interactions in a single representation. This is useful when there are different alternatives in a use case. However, this new notation still contains some ambiguities, as described in [Loidl97]. We refer to the previous and new versions of the MSC notation as MSC92 and MSC96, respectively. Sometimes the term "basic MSC" is used for a normal MSC to differentiate it from a HMSC.

A new visual notation for representing use cases, called Use Case Maps (UCM), has been proposed by Buhr [Buhr96]. In this notation, a visual link is drawn across the components of a system in the order in which they interact, and responsibilities are assigned to certain points on this link to describe the actions. However, UCM is an informal notation and there is ongoing work by Amyot [Amyot97] to develop a formal representation.

There are very few tools which relate to the TTCN or MSC notations. Two of the most well known are ObjectGEODE from Verilog [GEODE95] and Telelogic Tau (τ) from Telelogic [Tele97]. ObjectGEODE is a complete toolset for the development of real-time distributed applications in an object-oriented environment. It uses MSCs to describe scenarios and develop test cases. Telelogic Tau is a combination of two tools, namely ITEX 3.1 which uses TTCN and SDT 3.1 which uses the Specification Description Language (SDL), a language intended for the specification of complex, event-driven and real-time applications. Telelogic Tau also uses MSCs and the Object Modeling Technique (OMT) [Rumb91] which is used in UML. MSCs are used to describe use cases and TTCN is used to generate test cases. The tool allows automatic conversion between MSC and SDL.

There is ongoing work by Grabowski [Grab94] to develop a method for generating complete TTCN test cases from MSCs. This is useful for combining several related MSCs into a single representation.

In our work we discuss about message interactions and there are some overlapping concepts with work done by Kirani [Kirani94] and Benhajla [Benhaj97] on object-oriented testing. We consider message interactions between actors and they consider message interactions between objects in object-oriented models.

We propose a new notation for capturing use cases from informal requirements called Use Case Trees (UCTs). In the past there was work done by Logrippo and Guillemot [Log89] to generate test suites from formal specifications expressed in LOTOS. They used the concept of trees as well.

To our knowledge, no other work has been done in using TTCN to describe use cases, nor to convert from TTCN to MSC.

2.6 Example of a System

For the purpose of relating some of the concepts used throughout this thesis to a concrete example, we will describe a simple fax system [Prob94]. This is the example which will be referred to, every time we talk about the fax system. In chapter 7 we will describe the system in more detail.

Our simple fax system consists of a user, a sending fax machine and a destination fax machine, as shown in figure 2.3. For simplicity reasons, we assume that the sending fax machine cannot receive any documents, and hence its name. Also, we do not consider the telephone network as a component in the system.

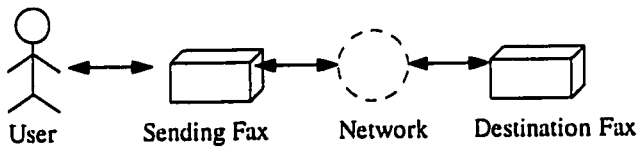


Figure 2.3

The following is a simple English description of the sequence of actions that takes place when a user wants to send a document :

The user inserts a document in the sending fax machine.
He then enters the destination number.
Finally he presses the GO button.
The sending fax sends a connection request to the destination fax machine.
The destination fax sends back a ring signal followed by an answer.
The sending fax then transmits the data.
After the transmission the sending fax displays the current time.

When a user wants to send a document, he may not be successful. If the destination fax machine is busy, the latter will send back a busy signal. Or it can happen that there is a problem with the destination fax and it does not send back an answer. In both cases, the process is aborted and an error message is displayed, followed by the time.

In this chapter we have introduced our current work and explained some basic concepts of software engineering. The subsequent chapters will describe in more detail our current work. We will start by describing ROOM and the ObjecTime toolset which will used later on in our case study.

Chapter 3

ROOM and the ObjecTime Toolset

In this chapter ROOM, a modeling language which allows incremental development of a software system is presented. This chapter is important since our case study is based on the ROOM development process. However, only the elements of ROOM which are used in the case study are described. For a complete description of ROOM please refer to [Selic94].

First an overview of ROOM is given along with the principles on which it is based. Then the two most important models of ROOM are presented, namely the structure model and the behaviour model. All systems are developed using these two models which complement each other. Finally the representation of MSCs in ROOM is given.

3.1 Overview of ROOM

ROOM, which stands for, Real-time Object-Oriented Modeling, is an object-oriented modeling language, primarily meant for developing distributed real-time systems. ROOM offers the ability to capture requirements, build a model and generate code from the model, all in a continuous process. Developing a system using ROOM is an incremental process. When a model is built, it can be compiled, run and tested, even at an early stage, and then further enhanced. This reduces the chance of getting flaws in a later stage of the development.

ROOM is graphically based. This helps to visualize and understand the modeling process better. The modeler does not have to bother about writing code, except those needed in the model itself, since code is automatically generated from the model. All he

has to know is how to use the graphical components, to model a system. Note that we use the term modeling rather than designing because a model is an incomplete system which can be compiled, run and tested, whereas a design cannot be.

ROOM is inherently object-oriented, which allows it to fully exploit the advantages of this significant new paradigm. All the object-oriented concepts are showed graphically in ROOM. Entities which are modeled in ROOM are reusable. This is more powerful than reusable code because it is at a higher level. This may save a lot of time if we have to model systems having common components.

The ROOM modeling language was devised to model systems that have certain characteristics of real-time systems. They are :

Timeliness - specific deadlines must be met. ROOM models execute in a time-efficient manner and are priority driven.

Dynamic internal structure - system components must be reconfigured as the environment changes. ROOM supports run-time creation/destruction of model components.

Reactiveness - the system must respond to events whose occurrence and order is unpredictable. ROOM models are event-driven.

Concurrency - the system must monitor and control multiple simultaneous activities. At any given time, multiple simultaneous activities can be taking place in the ROOM components.

Distribution - system components at multiple sites must cooperate. ROOM models contain independent components with their own structural and behavioural complexity but which can communicate between each other.

ROOM is organized around three main elements :

The operational approach - this eliminates discontinuities in the development process by using a single, integrated, formal set of notation to create a model, at whichever level of detail.

A phase-independent set of modeling abstractions - this eliminates discontinuities in the development process by bringing together the requirements, design and implementation representations.

The object paradigm - this represent a model as a group of interworking, reusable objects.

The ObjecTime toolset is the CASE tool which automates the ROOM methodology. It provides a modeling environment to support the ROOM language, including model capture and display, model analysis, and model execution.

3.2 The Structure Model

3.2.1 Actors and Actor Classes

In the object paradigm, objects are independent, concurrently active logical machines. ROOM refers to such logical machines as actors. Actors are created on the basis of actor class definitions. Actor classes act like templates for actors. An actor is the graphical object derived from an actor class definition and is represented as a square box. The boundary of the box represents the actor interface.

Let us consider the fax system example of section 2.6. One actor class that we can deduce from the requirements is "FaxMachine". Actors of "FaxMachine" can be named "SendingFaxMachine" and "DestinationFaxMachine". Figure 3.1 shows the actor class definition for FaxMachine as an entry on a list of such definitions and figure 3.2 shows the basic representation of the SendingFaxMachine in ROOM.

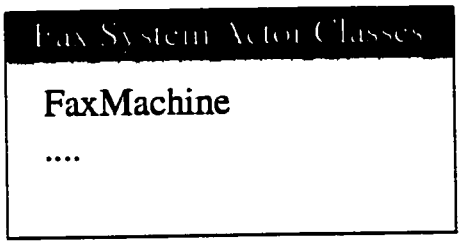


Figure 3.1

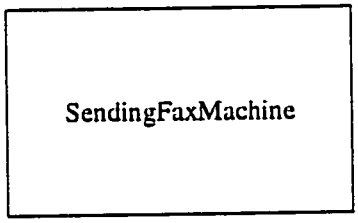


Figure 3.2

3.2.2 Messages, Protocol Classes and Ports

Actors communicate by sending and receiving messages. A message consists of a signal indicating the type of the message, a data object containing the values carried in the message, and a priority. In the fax system example, some messages which are exchanged between the SendingFaxMachine and the DestinationFaxMachine are, "ring", "answer", "data", and so on. There can be a very large number of messages exchanged and it is more appropriate to group together related messages under a single name. One such group is called a protocol class. The grouping of messages is left to the modeler's discretion. For our fax system we can, for example, group messages between the SendingFaxMachine and the DestinationFaxMachine under "CommSig", for communication signals, and the messages between the user and the SendingFaxMachine under "UserInput" and "UserOutput". In a protocol class, the direction, either in (received by the actor) or out (sent by the actor), of the set of messages has to be defined. Figure 3.3 shows some of the actor class and protocol class definitions of the fax system example. An expanded view of the protocol class CommSig is also shown.

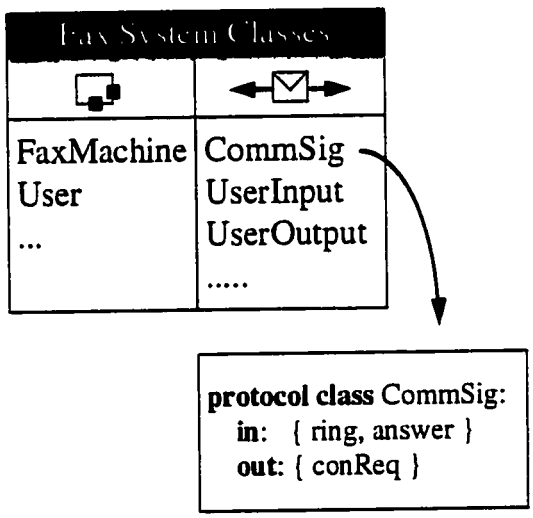


Figure 3.3

An actor's interface is defined by ports. Ports are derived from protocol classes. A port is a declaration that the set of messages defined by a protocol class forms an

integral part of the interface of an actor of a particular actor class. Any number of ports can refer to a single protocol class and the properties of the ports are governed by the protocol class definition. In ROOM, actors can contain other actors (we will discuss this in the section on hierarchical actor structure), the former being called the container actors and the latter being called the contained actors. Because ports are used to exchange messages between actors, and because the actors can be either container or contained, different types of ports are needed for communication. There are three types of ports in ROOM :

Interface relay ports - ports which are used to exchange messages between the outside world and the contained actor.
Interface end port - ports which are used to exchange messages between actors which are at the same level of containment.
Internal end ports - ports which allow a container actor to exchange messages with its contained actors.

Figure 3.4 shows the graphical representation of an end port and a relay port in ROOM. Note that internal end ports and interface end ports have the same representation.

End Port

Relay Port

Figure 3.4

Now let us have a look at the fax system to see the representation of the fax machine actor, along with its ports, in ROOM. Figure 3.5 shows the expanded structure of the actor class FaxMachine. The interface end ports are referred to the corresponding protocol classes.

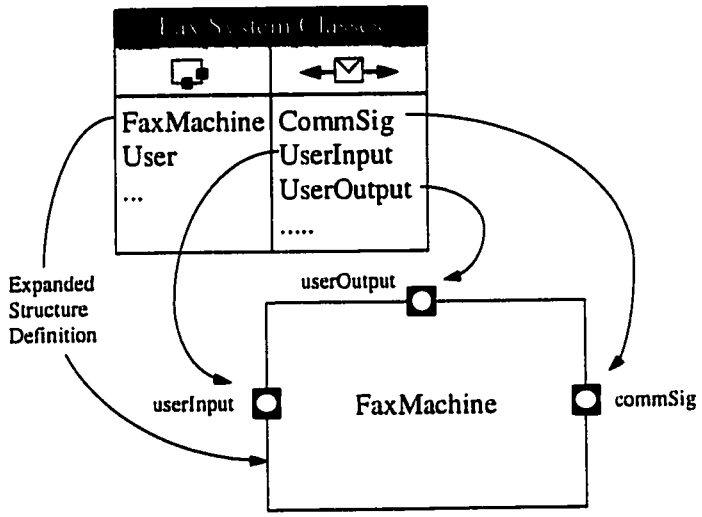


Figure 3.5

3.2.3 Conjugation and Bindings

For communication between two actors to take place, a link must be established between a port on one of the actors and a port on the other. The two ports to be linked must have the same protocol classes. A protocol class, as defined in ROOM has an imposed directionality (see figure 3.3). It defines the communication from the point of view of a particular actor, with one set of messages as incoming and another set as outgoing. When two actors communicate, however, the outgoing messages for one actor are the incoming messages for the other, and vice versa. ROOM deals with this complication by allowing a port to be conjugated. A conjugated port refers to a protocol class, but with the incoming and outgoing directions reversed. Both end ports and relay ports can be conjugated as shown in figure 3.6.

conjugated end port **conjugated relay port**

Figure 3.6

When two ports have been defined so as to permit communication, a link must be drawn to join the two ports, on the respective actors. This link is called a binding, and is simply shown as a line, in ROOM. Figure 3.7 shows two actors of the fax system, namely the User and the SendingFaxMachine, with their corresponding ports, and linked together through bindings.

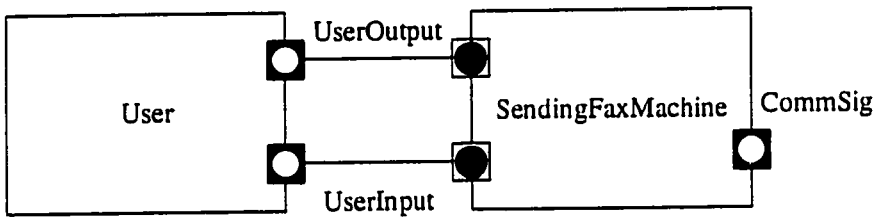


Figure 3.7

3.2.4 Hierarchical Actor Structure

An actor, as a logical machine, can have a larger scope; it can contain other actors as components. These components, in turn, may have other components of their own, to any desired depth of recursion. Actors which contain other actors are called container actors whereas the actors inside are called contained actors.

This notion of hierarchical actor structure is helpful in many ways. In the beginning, a modeler may want to model only a high level view of the system by describing the high level actors, without going in the structural details on these actors. He may then add more details to each component later on. Also this help to structure the system as a whole. Imagine that all the actors in a system were at the same level; this would represent an incomprehensible bunch of interlinked components.

Figure 3.8 shows a hierarchical structure of the fax system. The actor AFaxSytem contains the actors User, SendingFax and DestinationFax. This is a high level view of the fax system. We can go into further details by modeling the inner structure of a fax machine. For example we may add actors such as an input area to insert paper, a keypad to dial the destination number, a send button to send the document, and so on.

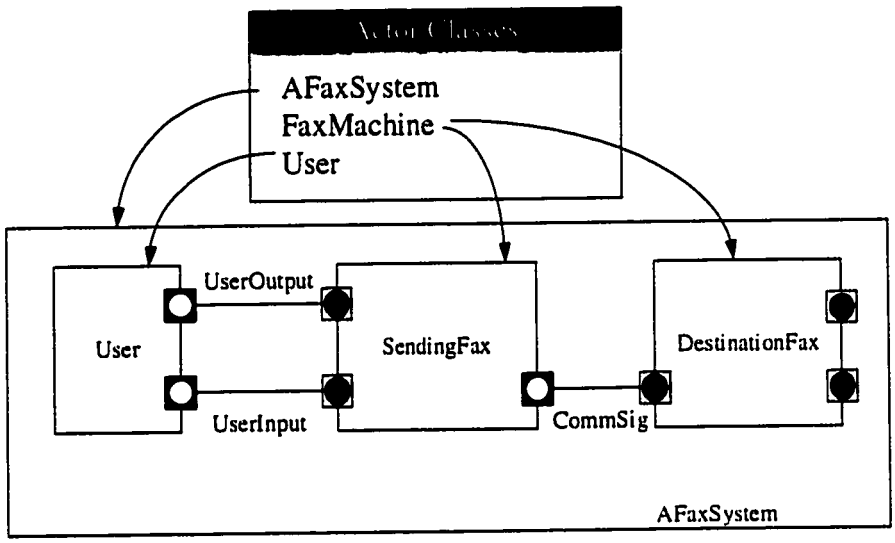


Figure 3.8

3.3 The Behaviour Model

3.3.1 ROOMcharts

In ROOM, an actor may have an internal operation, whereby messages are sent and received, and the actor changes from one state to another. This internal operation of an actor over time is called its behaviour. The behaviour of an actor is represented by an extended finite state machine called a ROOMchart. ROOMcharts, however, have some additional features over a conventional extended finite state machine, such as hierarchical states and entry/exit actions, which we will see in more detail later on.

A state is a static situation in which an actor finds itself and during which it can react to the arrival of a message. The arrival of a message is called an event. A transition is an action, triggered by an event, whereby an actor moves into another state. In ROOM, a state is represented by an elliptical shape and a transition is represented by an arrow, starting from one state and ending on another state. The behaviour of an actor is represented separately from its structure. Figure 3.9 shows the structure and behaviour of a fax machine as two distinct but complementary parts of the FaxMachine actor class.

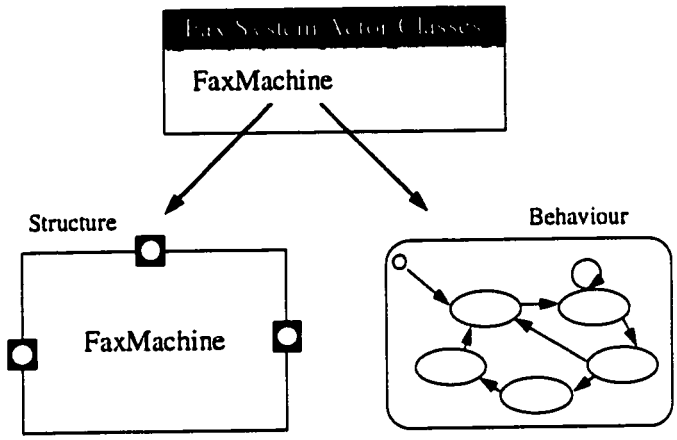


Figure 3.9

Note that in figure 3.9, the behaviour diagram is not representative of the behaviour of the fax machine, but instead shows an example of a graphical representation of a ROOMchart.

3.3.2 Triggers and Actions

Transitions in ROOMcharts are triggered by the arrival of messages through the interface of the actor whose behaviour is defined by the ROOMchart. Thus, each transition on a ROOMchart (except for transitions in the initial state) must have an attached trigger definition. Triggers are defined in the form of a list of port/message combinations, optionally followed by a call to a guard function. Figure 3.10 shows a ROOMchart representing the behaviour of the sending fax machine of the fax system. The transition "answer" is expanded to show the trigger definition.

From the trigger definition we can see that the transition whose name is "answer" is triggered by the incoming message answer through the port commSig.

The guard function is a Boolean function that must be evaluated when a message is received. The transition is taken only if the guard is true. By default the guard is assigned the value true.

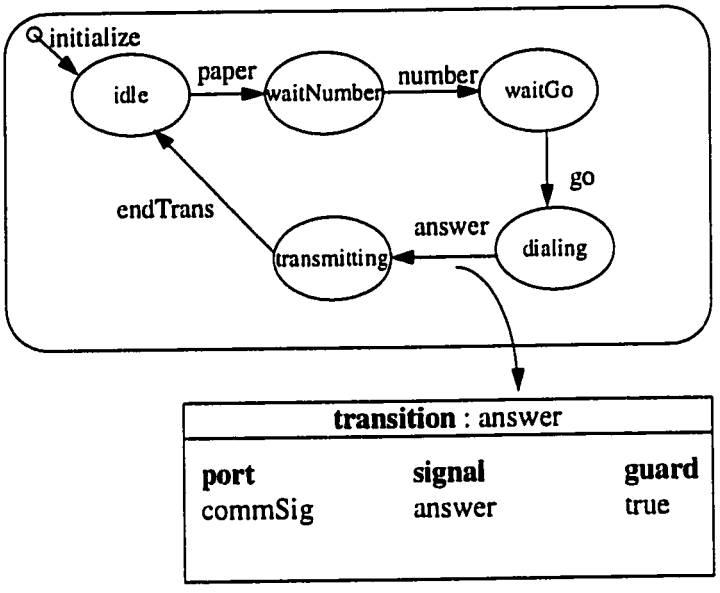


Figure 3.10

Various actions can be done by the actor, for example sending messages, changing data values, and so on. These actions are defined in the form of statements which are attached to the ROOMchart. A statement can be an elemental action, or a call to a function to perform a sequence of actions. An action, in the form of one or more statements, may be attached to

- A transition (including the initial transition)
- A state, as an entry action
- A state, as an exit action

These statements are executable instructions, written in a detail-level programming language incorporated in ROOM.

A transition which contains actions can be differentiated graphically from one which does not, as shown in figure 3.11. A transition with no actions has a unfilled tip, whereas one which contains actions has a filled tip.

Figure 3.11

An entry action is one which is done when a state is entered and an exit action is done when a state is exited. A state may have both entry and exit actions. The graphical representation of states with entry and exit actions is shown in figure 3.12.

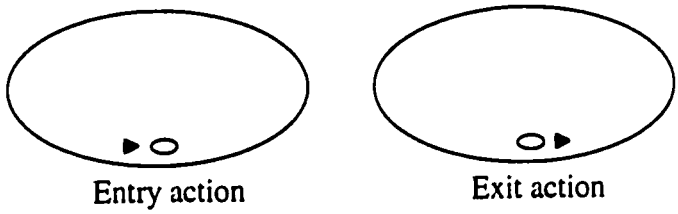


Figure 3.12

3.3.3 Hierarchical States

The ROOMcharts described in the previous sections have a single "flat" state machine. It is also possible to define hierarchical states, and much of the modeling power of ROOM comes from this capability. A hierarchical state is one which hides complexity, and which can be expanded to show the inside finite state machine. As with hierarchical actor structure, modeling is simplified, but this time by considering a high level view of the behaviour of an actor. The high level behaviour of an actor can be described at first, and further details added later on. A hierarchical state can allow group transitions. A group transition is represented by a transition which originates from a hierarchical state and which can be triggered from any state inside the hierarchical state.

Figure 3.13 shows a high level view of the behaviour of a fax machine. The hierarchical state "sending" is exploded to reveal the inner ROOMchart. The group transition "stop" can be triggered from any state inside the state "sending".

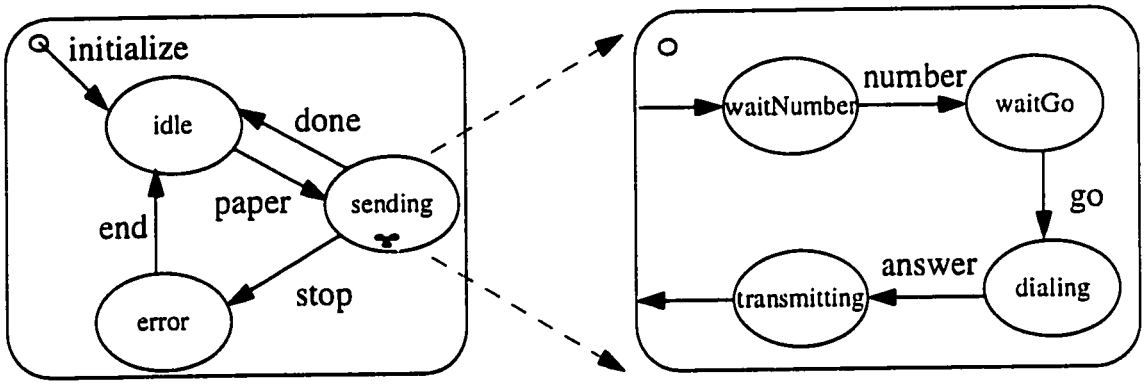


Figure 3.13

The ROOMchart on the left is the high level behaviour of the fax machine whereas the one on the right is the exploded state "sending". Note that this state has a little triangular shape below to indicate that this is a hierarchical state. When the message "paper" is received on the transition, the "sending" state is entered. The exploded ROOMchart is then considered. In our example, "stop" is a group transition because, at any state in the exploded ROOMchart, if a "stop" message is received, the state "sending" will be exited.

3.4 Message Sequence Charts

A Message Sequence Chart (MSC) is a representation of the flow of messages among model entities, and in the relative order in which they occur. It can be either graphical or textual. The standard for MSCs is described in the recommendation Z.120 of the ITU-T [MSC92] and is adopted by ROOM.

A graphical representation of an MSC is an easy way to describe and understand the message flow among the entities of a model. For this reason, MSCs are used to describe scenarios and requirements. In ROOM, MSCs can be captured as a form of requirement at the modeling phase, and can also be generated at execution, from the model, in both the textual and graphical forms.

Consider the fax system example and the scenario "SendOneFax". Figure 3.14 shows the message interactions between the User, the SendingFax and the DestinationFax, in the graphical form of a MSC.

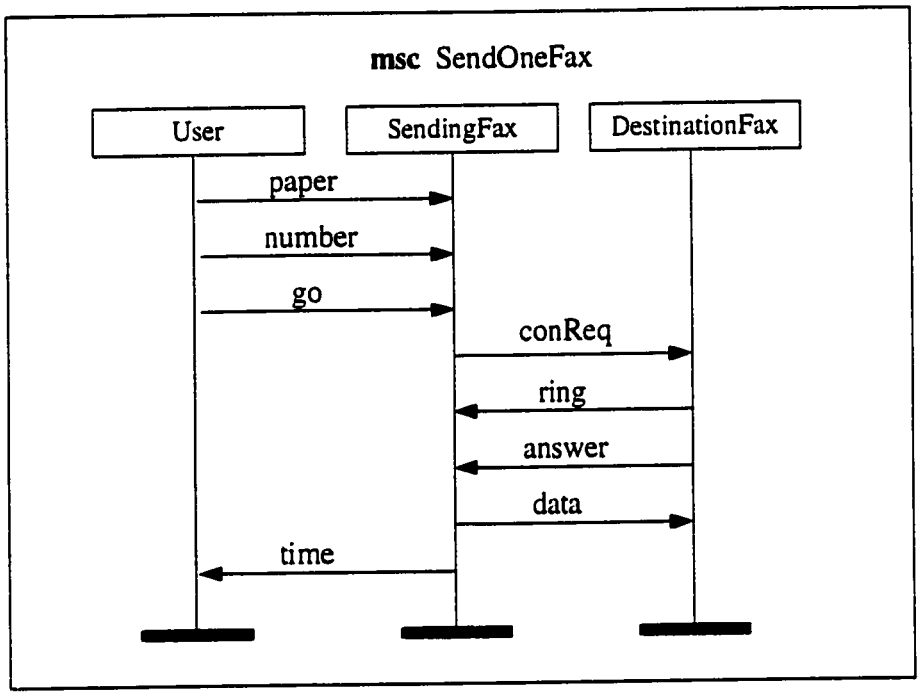


Figure 3.14

The arrows indicate the direction of flow of the messages. The vertical lines indicate the entities. In the diagram, time increases vertically down. The outer box indicates the environment, which represent any entities not explicitly identified in the chart. An arrow from, or to the outer box indicates that there is message interactions between the environment and the entity in the MSC.

The same example of message interactions is shown, in the textual form of a MSC, in figure 3.15. In the textual form, all the entities are described with their incoming and outgoing messages in the order in which they occur. The textual form, although less appealing than the graphical form, is easier to manipulate, and hence its widespread use for work concerning MSCs.

```
MSC SendOneFax;
    INSTANCE User;
        OUT paper TO SendingFax;
        OUT number TO SendingFax;
        OUT go TO SendingFax;
        IN time FROM SendingFax;
    ENDINSTANCE;
    INSTANCE SendingFax;
        IN paper FROM User;
        IN number FROM User;
        IN go FROM User;
        OUT conReq TO DestinationFax;
        IN ring FROM DestinationFax;
        IN answer FROM DestinationFax;
        OUT data TO DestinationFax;
        OUT time TO User;
    ENDINSTANCE;
    INSTANCE DestinationFax;
        IN conReq FROM SendingFax;
        OUT ring TO SendingFax;
        OUT answer TO SendingFax;
        IN data FROM SendingFax;
    ENDINSTANCE;
ENDMSC;
```

Figure 3.15

MSCs are important in our validation strategy because they are used as inputs to drive a model. For this purpose, textual MSCs are used. However, MSCs can also be used to describe requirements, and in this case the graphical representation is better.

In this chapter the main constituents of the ROOM methodology have been presented. These are used in our case study, which is described in chapter 7. The way ROOM handles MSCs has also been described. The MSCs used to drive the model of a system have the same format as those used in ROOM. The driving MSCs are generated from Use Case Trees which are presented in the next chapter.

Chapter 4

Use Case Trees

In this chapter we present a new notation for writing use cases called Use Case Trees. This chapter is meant to give to the reader a detailed description of Use Case Trees along with several examples to understand the notation. The core of our work is based on Use Case Trees and hence the importance of this chapter. This chapter is complemented by Appendix A which presents the grammar of the notation.

At the end of the chapter, we do an assessment of the Use Case Tree notation and compare it with three existing notations for writing use cases. We also describe some present limitations of the notation.

4.1 Background

The Tree and Tabular Combined Notation (TTCN) [TTCN92] is a test notation, standardized by the International Organization for Standardization (ISO), and widely used for the specification of conformance tests for protocol implementations. TTCN is defined as consisting of a static declaration part, and a dynamic behaviour description part. The structure and syntax of the test notation is entirely directed towards describing desirable sequences of interactions between the entities involved in the conformance testing process.

As the name suggests, TTCN consists of a blend of two types of notations : a tree notation, which is used in the dynamic behaviour descriptions to describe events which can occur as alternative responses to a previous event, and a tabular component, which is used to simplify the representation of all static elements, such as data types, protocol data units formats, and so on. TTCN is available in two formats, a graphical form denoted by

TTCN.GR, and a machine processable form denoted by TTCN.MP. TTCN.GR is intended to be human readable and understandable. TTCN.MP is intended to be a packed format representation to allow more efficient storage and for electronic transfer of test suites.

In our work we use a subset of the dynamic behaviour description of TTCN (defined in a tree notation) and the graphical representation to produce a notation which can be used to describe use cases. We named this notation Use Case Trees (UCT). We use the term Use Case Tree to denote both the notation and the tree representation. For the purpose of this work, UCT will refer to a subset of the dynamic part of TTCN. The static part of TTCN, including the tabular representation of data, will be analyzed in the future.

4.2 Objectives of Use Case Trees

4.2.1 Representing Use Cases

UCT helps to represent use cases in a tree notation. In this notation we can group the basic course and all the alternative courses associated with a use case in a single, easy to understand, and readable tree type structure. The graphical representation helps to visualize which courses can be taken at various moments in time. Some examples will be shown in the coming sections, after the basics of representing a use case in UCT have been explained.

Right now there is no standard notation for representing use cases. Most people will use the notation which they think is the best. Some existing notations [Buhr96, Jacob92], which describe use cases either textually or graphically, are used only to describe the behaviour of a software system. But with UCT, we can do more than describing behaviour, we can use this description to validate the system under construction. UCT has a formal way to represent use cases, with a well defined grammar [Appendix A], derived from the TTCN grammar. This makes use case writing unambiguous.

Present techniques used for representing use cases lack a compact representation of a single use case. Most of the time the basic path and the alternative paths of the use

case will be represented separately. This may lead to difficulties in reading and understanding since most of the time a use case may have different alternative paths and we may need a few sheets of paper to describe all of them one by one. We think that we should be able to describe a use case fully on a single sheet of paper for readability. It is much easier to go through a single page to see the complete use case rather than going through a bunch of sheets. Also, the bigger the representation of a use case the more space it will take for storage.

4.2.2 Compacting Scenarios

We can describe a system by using scenarios rather than use cases. Some scenarios may have common paths or even a preamble which is common to all of them. In such a case we can use UCT to compact them. The scenarios can be represented in a single use case with different alternative paths, depending on the number of scenarios. Let us consider the fax system as an example. Two possible scenarios when we want to send a document are, "sending a document normally", and "sending a document and the destination fax is busy". We can compact these two scenarios in a single use case which can be called "sending a document". When we send a document there is a sequence of initiating events which can be; the user inserts the document, the user dials the number, the user hits the go button, and the fax sends a connection request. This sequence of events can be a preamble in the use case "sending a document". Then we can add the sequence of events corresponding to the two scenarios as alternatives in the use case as shown in Figure 4.1.

We see that the structure looks like a tree, with a sequence of events (preamble) at the root, and two branches for the two alternatives. We do not have to repeat the preamble part for each scenario and this saves on time and space without decreasing the readability. UCT uses the same notion of tree representation, but does not show nodes and branches graphically. The concept of nodes and branches is hidden in the way the use case is written. The next section will explain the representation.

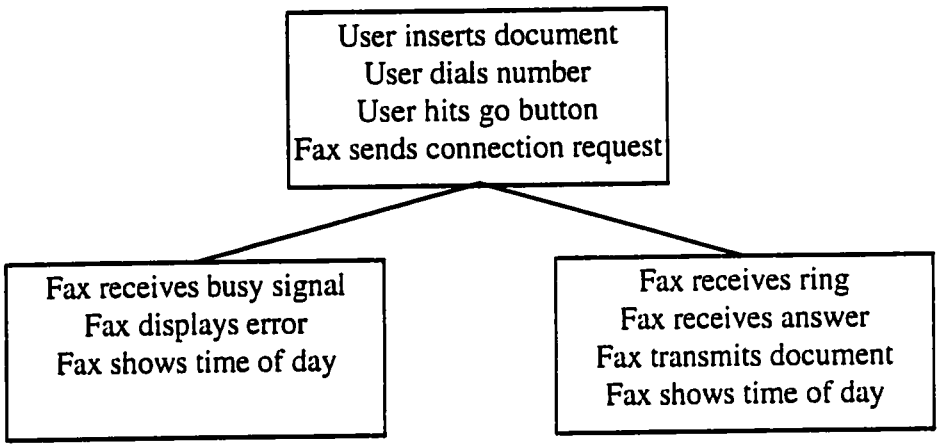


Figure 4.1

4.3 Description of Use Case Trees

4.3.1 The Tree Representation

A use case is basically a collection of sequence of events. There may or may not be any preamble or alternatives. UCT uses a tree representation to show the sequence of events. The following example will help to explain the representation better.

Suppose we have a use case which starts with an event A, followed by two alternative events, B and K. After event B we have three alternative events namely C, D and E. After event E we have two alternative events, namely I and J, and after event C we have another two alternative events, namely F and G. Finally after event G we have event H. This sequence of events can be represented in a tree structure as shown in Figure 4.2.

Figure 4.3 shows the UCT representation of this use case. Each event is placed on a new line called a behaviour line. The time axis is as shown in the figure. To show that event B takes place after event A, the former is indented using the characters "^^". We use two "^" to show one level of indentation for two main reasons. Firstly to make it more readable and secondly to reduce errors of mistype. Any odd value of "^" is an error. Indentation is a key factor in this representation and has to be placed correctly. Each new event is indented once from left to right, with respect to its predecessor.

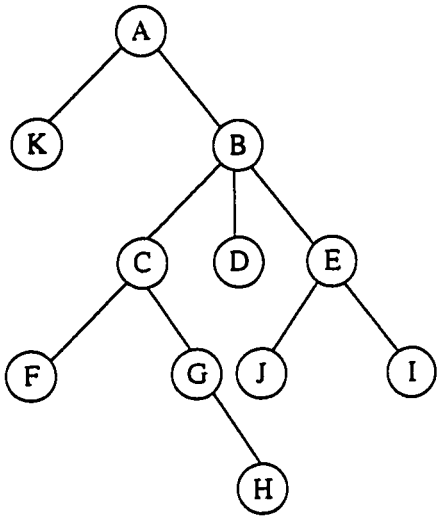


Figure 4.2

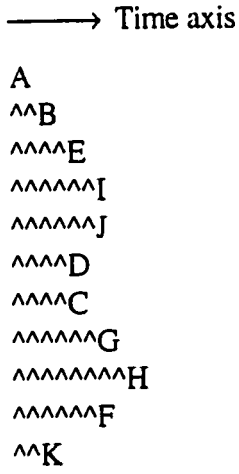


Figure 4.3

Events at the same level of indentation are called alternatives. At any moment in time only one alternative is executed. For example, after event B, only one of events C, D and E will be executed.

From Figure 4.2, if we start from the event A (the root of the tree), there are a number of paths that can be taken to reach the different leaves. Each path will be a scenario for the use case. In fact we have six different scenarios in our example, namely "A B E I", "A B E J", "A B D", "A B C G H", "A B C F" and "A K". We can see the same sequences if we look at Figure 4.3. From event A we go down to the right until we cannot go any further. We obtain the sequence "A B E I". Then we go back to the beginning and restarting from event A we go down to the right, until we reach a node where we have to choose another alternative. This time we get the sequence "A B E J". Each time we reach a leaf, the nodes we visited will produce a complete scenario. By repeating this process we finally get all the possible paths, or scenarios, in the tree.

4.3.2 The General Structure

A use case tree has a well defined structure. First of all we write the TreeIdentifier, which may represent the name of the use case. It can be a meaningful, but single word. We have the option of having a formal parameter list after the TreeIdentifier, for exchange of information between UCTs. One UCT can call another UCT, just like a procedure call, in any programming language. We will refer to a calling UCT as a MainTree and a called UCT as a SubTree. Note that both of them have the same structure. We will discuss this in more detail in the section on construction mechanisms. Then we write the behaviour description in a sequence of behaviour lines with the appropriate indentation, as in Figure 4.3. Finally we write the characters "%%" to show the end of the tree structure. The general structure of a UCT is as follows :

<TreeIdentifier> [FormalParameterList]
Sequence of behaviour lines in a tree structure
%%

4.3.3 The Behaviour Description

In UCT we have four kinds of elements which may be used to fully describe the behaviour of a use case. They are :

- communication events
- timer events
- assignments, operations and qualifiers
- construction mechanisms

A use case can be described using a combination of these elements. All of them are described in the UCT grammar [Appendix A]. In the following sections we will describe each element in more detail.

4.3.3.1 Communication Events

Communication events are messages which are exchanged between entities in a system. In a system, modeled using ObjecTime for example, these are messages which are

exchanged between actors. In UCT, a use case will only describe communication events which are related to the actor which is under test, or AUT. The AUT may be connected to other actors via different ports. These ports, on the AUT, are the points through which messages are sent or received. We can see a port as being a point of control and observation, or PCO, because we can control the AUT by sending any desired message through its ports, and, at the same time, observe its behaviour, by analyzing which messages are sent out. If we are to describe the behaviour of a AUT, we should specify which messages are sent and received though which PCO. Figure 4.4 shows the fax system, with the SendingFax as the AUT. The ports UserInput, UserOutput and CommSig are the PCOs of the SendingFax. The description of a use case in UCT is always done with respect to the AUT.

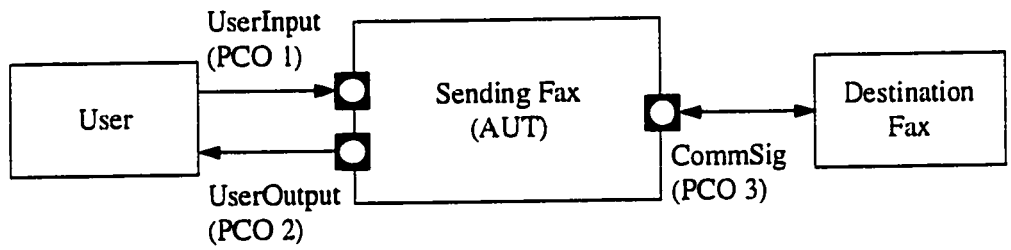


Figure 4.4

There are three basic communication events in UCT :

<PCO>!<message>	The AUT sends <message> to <PCO>
<PCO>?<message>	The AUT receives <message> from <PCO>
<PCO>?OTHERWISE	The AUT receives any message from <PCO>

The exclamation mark (!) represents a send and the question mark (?) represents a receive. The predefined event OTHERWISE is the UCT mechanism for dealing with unforeseen events in a controlled way. OTHERWISE is used to denote that the AUT shall accept any incoming message from the PCO which has not previously matched one

of the alternatives to the OTHERWISE. It is usually placed at the end of a list of alternatives since it will always be executed.

The tree notation and the communication events are combined to describe the use case "sending a document", as shown in Figure 4.5. This use case describes how the SendingFaxMachine, the actor under test, will communicate with other actors via its ports, the points of control and observation. We see how the two scenarios ("sending a document normally" and "sending a document and the DestinationFax is busy") are compacted in a single use case. The messages "ring" and "busy" are alternatives, meaning that only one can be executed at a time. The preamble of the use case is written only once and the different scenarios can be easily read from the representation. We see clearly which messages are going through the PCOs and in what order. Also the names of the PCOs and messages are meaningful, which add to clarity.

```
UseCaseSendingADocument
UserInput?document
^^UserInput?number
^^^^UserInput?go
^^^^^^CommSig!connectionRequest
^^^^^^^^CommSig?ring
^^^^^^^^^^CommSig?answer
^^^^^^^^^^^^CommSig!data
^^^^^^^^^^^^^^UserOutput!timeOfDay
^^^^^^^^CommSig?busy
^^^^^^^^^^UserOutput!error
^^^^^^^^^^^^UserOutput!timeOfDay
%%
```

Figure 4.5

4.3.3.2 Timer Events

Time is an important aspect when dealing with real time systems. In UCT it is possible to use timers, operations on timers, and to express expiration of a timer with a timeout event in a behaviour line. A timer can be in three states; it is either inactive, running or expired. In a use case all declared timers are inactive at first.

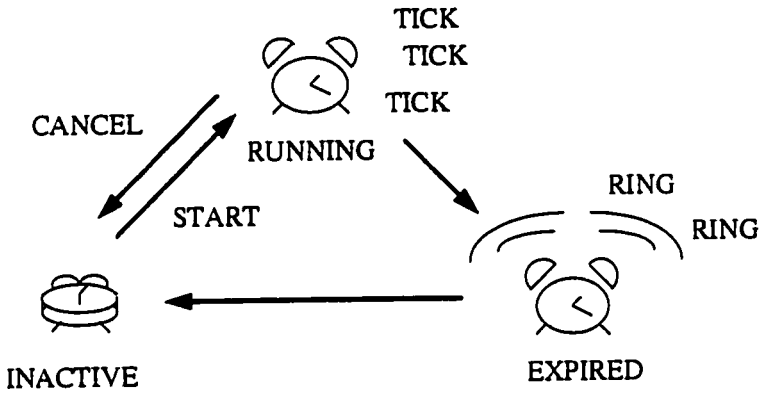


Figure 4.6

Figure 4.6 shows how a timer can transfer from state to state. A change of state is a result of manipulation of timers, with the following operations in UCT :

START <TimerId> <TimerValue>	activates a timer, it starts running counting down from its declared duration to zero.
CANCEL <TimerId>	inactivates a running timer

The remaining time of a timer can be consulted with the operation :

READTIMER <TimerId> <VarId>	the result of this operation is that if the timer indicated by <TimerId> is running, the variable <VarId> gets the value of the number of time units indicated by the timer. If the timer is inactive or expired, the variable will contain the value zero.

In a use case we can use a timeout event in a behaviour line to express the expiration of a timer.

?TIMEOUT <TimerId>	this statement checks if the specified timer has expired. If so, the execution of the use case is proceeded at the next level on indentation.

As an example, let us consider the fax system. If after an error message there should be a five seconds wait before the time is displayed, then part of the behaviour description would look like this :

```
UserOutput!error
^^START Timer1 (5)
^^^^?TIMEOUT Timer1
^^^^^^UserOutput!timeOfDay
```

A timer named Timer1 starts a countdown from 5 seconds after the message error is sent through the port UserOutput. The message timeOfDay can be sent only after Timer1 expires.

4.3.3.3 Assignments, Operations and Qualifiers

Sometimes it is necessary to retain some information about previous events in order to describe a use case properly. In UCT, variables can be used to store this information. It is good practice to give variables meaningful names to increase readability and reduce ambiguity. A value can be assigned to a variable by means of an assignment :

<VarIdentifier> := <Expression>

Assignments are stated in the behaviour description of a use case, either following a communication event or on a line of their own. In case there are two or more assignments on a line, they are separated by commas. A list of assignments is always surrounded by parentheses. The expression at the right hand side of the assignment can be a single value or a combination of variables and values linked by operations.
In UCT the following predefined operations can be used :

"+", "-"	for additive operations
"*", "/"	for multiplicative operations
"=", "<", ">", "<=", ">=", "<>"	for relational operations
"AND", "OR", "NOT"	for Boolean operations

Using these operations, we can write expressions in assignments such as :

Counter := Counter + 1
Continue := (Value < MAXVALUE)
Continue := ((Value < MAXVALUE) AND (NOT (Finish)))

An expression will always result in a numerical value or a Boolean value. A qualifier is a Boolean expression surrounded by square brackets, that can be placed in the behaviour description of a use case. A qualifier has a high priority and , if it evaluates to FALSE, the whole line, and subsequent lines at which the qualifier is stated, are not taken into account.

An event may be associated with an assignment, a qualifier or both. If an event is associated with an assignment, the assignment is executed only if the event matches. If an event is associated with a qualifier, the event may match only if the qualifier evaluates to TRUE. If an event is associated with both, the event may match and the assignment is executed, only if the qualifier evaluates to TRUE.

Consider the fax system example. Let us assume that the user has to enter a seven digit number to make a call. We can use a qualifier on the behaviour line, after the AUT

receives the number, to say that if this condition is not satisfied, the subsequent events are aborted, as shown in Figure 4.7. If number is not a SevenDigitNumber then "go" cannot be received. Note that the qualifier is indented to show that it is evaluated after the AUT receives "number".

```
UseCaseSendingAFax
UserInput?document
^^UserInput?number
^^^^[number = SevenDigitNumber]
^^^^^^UserInput?go
^^^^^^^^CommSig!connectionRequest
```

Figure 4.7

4.3.3.4 Construction Mechanisms

In UCT there are two construction mechanisms that can be used to build up the relevant parts of the behaviour tree. They are the Attach construct, for attaching use case trees, and the Repeat construct, for building loops. They can help to save space in the representation of large behaviour trees and to structure use cases properly. Let us first see the Attach construct.

When writing a use case, we may find that a sequence of events is repetitive at various positions in the main use case tree, which we will call the MainTree. In UCT we can combine this sequence of events in another use case tree, which we will call a SubTree. The SubTree will then be written after the MainTree. To access the SubTree, we write an Attach construct at the right position in the MainTree. The syntax for writing an Attach construct is as follows :

```
+ <TreeIdentifier> [ ActualParameterList ]
```

The Attach construct is placed alone on a behaviour line and at the correct indentation. "Attach" does not appear explicitly in the notation but is represented by the '+' symbol. The TreeIdentifier is the name that will correspond to the SubTree. The ActualParameterList is optional and is a list of parameters which are used to pass information from the MainTree to the SubTree, and vice versa. The parameters can be actual values or variables.

There is a set of rules for attaching SubTrees at specified points in the tree structure. The rules ensure that the correct equivalent expansion of the full tree is preserved. When attaching a SubTree, the indentation of the next event, right after the Attach construct, will determine the expansion of the full tree.

- If the indentation of the next event is more than that of the Attach construct, then the next event (and all the subsequent events which have a higher indentation than the Attach construct) will be attached to each leaf of the SubTree after the expansion. An example is shown in Figure 4.8. We see that event C, after the Attach construct, is attached after Q and R, the leaves of the SubTree.

MainTree	SubTree	MainTree (Expanded)
A	P	A
^^B	^^Q	^^B
^^^^+SubTree	^^R	^^^^P
^^^^^^C		^^^^^^Q
^^^^D		^^^^^^^^C
		^^^^^^R
		^^^^^^^^C
		^^^^D

Figure 4.8

- If the indentation of the next event is less or equal to that of the Attach construct, then the SubTree is attached as is, at the proper place. An example is shown in Figure 4.9.

MainTree	SubTree	MainTree (Expanded)
A	P	A
^^+SubTree	^^Q	^^P
^^B		^^^^Q
^^^^C		^^B
^^^^+SubTree		^^^^C
^^D		^^^^P
		^^^^^^Q
		^^D

Figure 4.9

A MainTree may contain more than one Attach constructs. In UCT, SubTrees can also contain Attach constructs. As in MainTrees, this is helpful when there is repetition in a SubTree, or in different SubTrees since more than one SubTrees can call the same SubTree. The example of Figure 4.10 shows how a MainTree, which calls two SubTrees (S1 and S2), which in turn call one SubTree (S3), is used to represent the same information as a fully expanded tree. Note that event B is placed after each leaf of SubTree S1 (only S3) and subsequently after each leaf of SubTree S3 (only W).

MainTree	S1	S2	S3	MainTree (Expanded)
A	P	Q	V	A
^^+S1	^^+S3	^^R	^^W	^^P
^^^^B		+S3		^^^^V
^^+S2				^^^^^^W
				^^^^^^^^B
				^^Q
				^^^^R
				^^V
				^^^^W

Figure 4.10

As stated earlier, information can be passed from a calling use case tree to the SubTree, and vice versa, through parameters. There should be the same number of actual and formal parameters in an Attach construct and the corresponding SubTree. In Figure 4.11, A, V and W are events, X is an actual parameter, and P and Q are formal parameters. When SubTree S1 is called from the MainTree, the value 2 is passed through parameter Q, to the SubTree. If event W is successful, P is assigned the value contained in Q, which is 2. We then return to the calling tree and X now has the same value as P, which is 2.

MainTree	S1(P,Q : INTEGER)
A	V
^^+S1(X,2)	^^W (P := Q)

Figure 4.11

The second construction mechanism is the Repeat construct. The Repeat construct is used to build loops in a use case tree. Sometimes we have an event or a group of events which is repetitive in a use case. The best way to compact it is to use a looping notation, with a condition of exit. The Repeat construct has the following syntax :

REPEAT <TreeIdentifier> [ActualParameterList] UNTIL Qualifier

The Repeat construct will keep on calling a SubTree until the Qualifier evaluates to TRUE. The SubTree will be called at least once. After each call, the Qualifier will be evaluated. The example of Figure 4.12 shows how the Repeat construct can be used.

(Counter := 0)	S1(X : INTEGER)
A	^^B (X := X + 1)
^^REPEAT S1(Counter) UNTIL [Counter = 3]	

Figure 4.12

A and B are events, and Counter is a variable initialized to 0. After event A is successful, SubTree S1 will be called. If event B is successful, X will be increased by one and the SubTree will be exited. Then Counter will be evaluated and if it is not equal to 3, S1 will be called again. In our example, S1 will be called three times.

4.3.4 Representing the Type of Scenario

In UCT we have the option of associating a type to each scenario which makes up a use case. The type of scenario, in the context of UCT, corresponds either to a scenario describing the normal behaviour of a system, referred to as a *normal scenario*, or to a scenario describing a possible error condition, referred to as an *exceptional scenario*. This is comparable to Jacobson's basic and alternative courses. An error condition here is a behaviour which is different from the normal expected behaviour but which does not necessarily mean that the system is behaving badly. If we consider the use case SendingAFax for example, an error condition can be the reception of a busy signal and the fax is not transmitted.

So in a use case we have one normal scenario and possibly, many exceptional scenarios. Exceptional scenarios may be classified according to different criteria such as their rate of occurrence, their consequences on the system, the cost associated for testing them, and so on. In our work, for simplicity reasons, we consider only two types of exceptional scenarios, namely *high risk scenarios* and *low risk scenarios*. A high risk scenario is one describing an error condition which will be costly if not properly tested. A low risk scenario, on the other hand, is one describing an error condition which will have little consequences if this situation occurs. Thus, it may or may not be tested depending on the time available or the cost of testing. However, the choice of risks is subjective and depends on the importance of the scenarios. We represent the three types of scenarios that we are considering in our work as follows :

- Normal scenarios — represented as N in UCT
- Low risk scenarios — represented as L in UCT
- High risk scenarios — represented as H in UCT

In UCT, we show the type of scenario, on the right hand side of a leaf in the use case tree. For use cases which are described using a combination of use case trees, the type of scenario is usually associated in the MainTree. If the MainTree has to be reused as a SubTree, the type of scenario can be removed, since it is optional, and if needed, it can be associated to the higher level use case tree.

Figure 4.13 shows how we can associate a type to the two scenarios of the use case "SendingAFax". This example, at the same time, shows how the use case can be structured using Attach constructs. The MainTree consists of three Attach constructs. The first one, as the name says, is a preamble to the other two Attach constructs in the use case. NormalTransmission and DestinationFaxBusy are alternatives which are done after the preamble and at the same time are leaves in the use case and consequently, a type of scenario can be associated to each of them. NormalTransmission is the normal transmission process and is a normal scenario, denoted by the letter N. DestinationFaxBusy is an exceptional scenario, with a high risk, and is denoted by the letter H. This scenario is a high risk one because this situation happens very often and has to be tested so that the system does not crash whenever the destination fax is busy. The three Attach constructs are described as SubTrees below the MainTree.

```
UseCaseSendingAFax
+Preamble
^^+NormalTransmission      N
^^+DestinationFaxBusy      H
%%

Preamble
UserInput?document
^^UserInput?number
^^^^UserInput?go
^^^^^^CommSig!connectionRequest
%%
```

NormalTransmission

CommSig?ring

^^CommSig?answer

^^^^CommSig!data

^^^^^^UserOutput!timeOfDay

%%

DestinationFaxBusy

CommSig?busy

^^UserOutput!error

^^^^UserOutput!timeOfDay

%%

Figure 4.13

4.4 Assessment of the UCT Notation

Our new notation is powerful, and at the same time simple to write and understand. It is formal and hence will help to reduce ambiguities in use case writing. It can be used both for describing systems and validating them. Unlike the common consideration of a use case being the description of a user's actions on a system, and vice versa, we went a bit further by saying that a use case can also represent the internal behaviour of a system. This broadened our view of a use case, and now we can think of high level use cases, which describe a higher level behaviour, and refinement of the high level use cases, which can describe more detailed behaviour. Our notation offers this possibility by allowing us to describe a system using a single high level use case tree at first, and then attaching other use case trees to the first use case tree to show more details. The number of attachments is unlimited and the best is that a use case tree which is attached can also call other use case trees, and these in turn can call other use case trees, to an indefinite number of recursions.

We think that UCT is a very powerful notation because it allows more than one scenarios to be grouped together in a single representation. However, because we cannot run different scenarios at the same time on a single entity, we have to ungroup these scenarios to produce individual groups of message interactions, better represented by basic MSCs (MSC92), for the validation process. For this purpose we built a tool which accepts use cases written in UCT and automatically generates the corresponding MSCs, in the Z.120 format, as described in chapter 6. The tool also provided a means of checking the correctness of the UCT grammar presented in Appendix A.

Now we will show how the UCT notation compares with some existing notations for describing use cases, namely Jacobson's prose description, Buhr's use case maps, and finally, the MSC96 Z.120 recommendation. To do so, we will consider the UCT representation of the use case "SendingAFax" of figure 4.13, which shows only two scenarios, namely "NormalTransmission" and "DestinationBusy".

4.4.1 Examples of Existing Notations

UML notation

The UML notation uses Jacobson's notation to describe use cases. A use case diagram graphically shows the relationship among actors and use cases within a system. An example for our chosen use case is shown in figure 4.14. A use case is shown as an ellipse containing the name of the use case. The actors interacting with the system are shown using "stick man" figures. The participation of an actor in a use case is shown by connecting the actor symbol to the use case symbol by a solid line.

The use cases are described using Jacobson's prose description which describe only the interactions between the user and the system, and vice versa. Nothing is known about the internal message interactions of the sending fax machine.

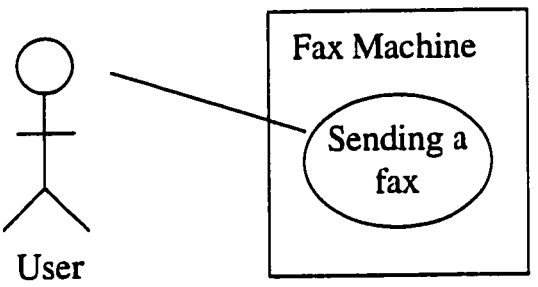


Figure 4.14

The description of the use case in UML is as follows :

Basic course - *Normal transmission*

When a user wants to send a document he first inserts a document in the sending fax machine. He then dials the destination number and finally, presses the GO button on the sending fax machine. When transmission is complete, he sees the current time on the display.

Alternative course - *Destination busy*

In case the destination fax machine is busy, the user will see a busy message on the display for five seconds, followed by the current time.

In UML, collaboration among objects can be shown using collaboration diagrams, derived from Object Modeling Diagrams (OMD), a contribution of Booch [Booch91]. Collaboration diagrams can be used to express simple scenarios graphically. The objects, or actors, are clearly visible along with the message interactions. However, in collaboration diagrams the explicit sequence of messages is not shown. They are used mainly to show the relationship among objects and are meant for understanding all the message interactions between them. Although collaboration diagrams and message sequence charts express the same information, they show it in different ways. In message sequence charts we see the explicit sequence of messages.

Buhr's Use Case Maps

There has been extensive work done by Buhr and Casselman [Buhr96] to develop a new notation called Use Case Maps (UCM) which describes use cases graphically. A use case map is a visual trace drawn across the entities of a system to describe the interaction patterns between them, and along which responsibilities can be added. This notation helps people express and reason about a system's behaviour.

Use Case Maps is a design notation enabling high-level designs as well as low-level designs. It bridges a modeling gap between requirements and design. Use Case Maps fill a gap in the suite of design models by providing a way of representing large-grained behaviour patterns as first-class abstractions above the level of realisation details. Use Case Maps offer something new in relation to architecture. They provide a behavioural framework for making architectural decisions at a high level of design, and also for characterising behaviour at the architectural level once the architecture is decided. Use Case Maps also provide a new technique for capturing large-grained behaviour patterns as concrete work products that may be saved, manipulated, extended, and reused to guide implementation, maintenance, and evolution.

The use case map approach comes at system modeling from an entirely different angle than, for example, state machines models or Petri net models, to name just two of many types of executable or mathematical models of systems. One use of such techniques is to express behaviour requirements for systems in a precise and relatively complete way, viewing the system as a black box. Among other things, this enables complex requirements to be checked by automation techniques, to help spot mistakes before they are built into implementations. The problem with such approaches from a design perspective is that they do not provide a progressive path to resolve high level design issues. Use Case Maps provide a solution by allowing designers to do informal modeling at a high level of abstraction and allowing for easier discussions among them to resolve the issues.

A more detailed description of use case maps is beyond the scope of this thesis. For more information please refer to [Buhr96].

In figure 4.15 we give a high-level description of the use case "SendingAFax" using the use case map notation. The starting point is shown with a black dot and the ending point with a vertical line. In our example, path a represents the "busy" scenario and path b represents the "normal" scenario.

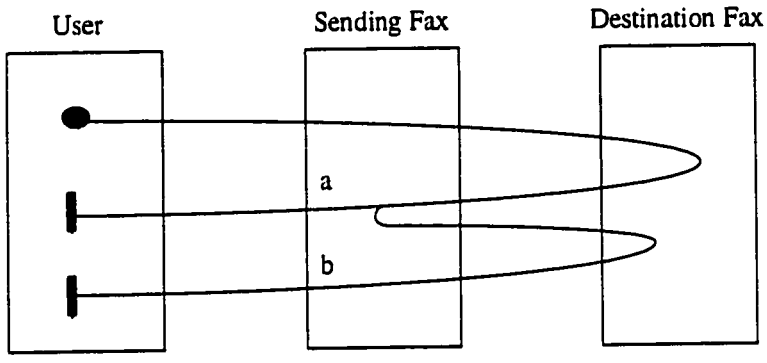


Figure 4.15

For a successful transmission of a fax, the user first has to interact with the Sending machine. The latter will then interact with the Destination machine which will do the same in return. Note that we do not have to know about which messages are exchanged at this level. Finally the Sending machine will interact with the user to inform him of a successful transmission. This is shown by path b in figure 4.15. If the Destination machine is busy then path a will be done.

So, by using this very simple technique, designers can describe the temporal interactions among entities of a system. Note that along the paths, responsibilities can be assigned for the various entities.

MSC 96

During the past couple of years there has been extensive work done by a number of people to come up with a notation which is more powerful and more effective than MSC92. The new notation is known as MSC96 and has already been standardized by ITU-T [MSC96]. Main emphasis has been put on the development of new structural concepts, namely generalised ordering, new formulation of instance decomposition, inline expressions, MSC references and High-level Message Sequence Charts (HMSCs).

Generalised ordering is used to describe the temporal order of two events, in case this cannot be deduced from the ordering imposed by the instances and messages. By means of instance decomposition, a refining MSC may be attached to an instance, which describes the events of the decomposed instance on a more detailed level. Thus, instance decomposition determines the transition between different levels of abstraction. By means of inline expressions, composition of event structures may be defined inside an MSC. The composition operators refer to alternative and parallel composition, iteration, exception and optional regions. MSC references are used to refer to other MSCs of the MSC document. The MSC references are objects of the type given by the referenced MSC. MSC references may not only refer to a single MSC but also to MSC reference expressions and MSC references. High-level Message Sequence Charts provide a means of graphically defining how a set of MSCs can be combined. The composition of MSCs specified by HMSCs can be guarded by conditions in the HMSCs. This is very much like what is available in UCT.

MSC96 is very new and there are still a few adjustments to be made, as discussed in a few documents [Loidl97]. However, we believe that this notation will become more and more popular in the designer community because of its power and simplicity and also because of the present popularity of MSC92. A more detailed description of MSC96 is beyond the scope of this thesis. For more information about MSC96 please refer to [MSC96]. Next we will show how our use case can be represented using the MSC96 notation.

Figure 4.16 shows the HMSC graphical representation of the use case "SendingAFax". This HMSC has three MSC references namely "Preamble", "Normal" and "Busy". The textual representation of the HMSC is later described.

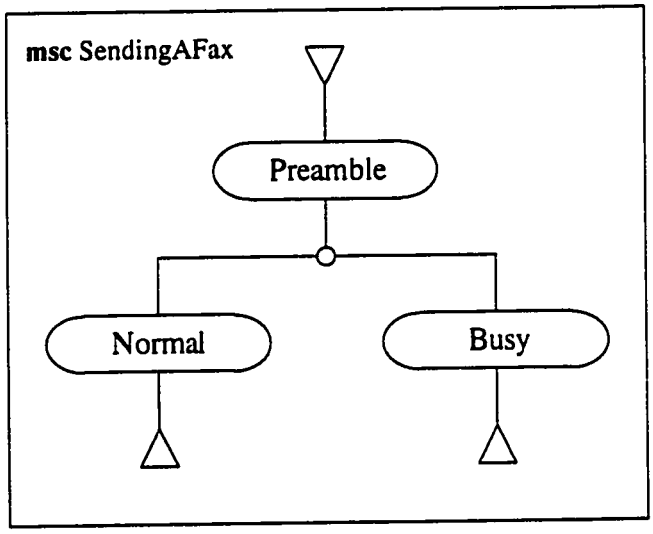


Figure 4.16

```
MSC SendingAFax;
EXPR L1;
        L1 : Preamble SEQ (L2 ALT L3);
        L2 : Normal SEQ (L4);
        L3 : Busy SEQ (L4);
        L4 : END;
ENDMSC;

MSC Preamble;
        INSTANCE  AUT;
                IN document FROM userInput;
                IN number FROM userInput;
                IN go FROM userInput;
                OUT connectionRequest TO commSig;
        ENDINSTANCE;
ENDMSC;
```

```
MSC Normal;
        INSTANCE AUT;
                IN ring FROM commSig;
                IN answer FROM commSig;
                OUT data TO commSig;
                OUT timeOfDay TO userOutput;
        ENDINSTANCE;
ENDMSC;

MSC Busy;
        INSTANCE AUT;
                IN busy FROM commSig;
                OUT error TO userOutput;
                OUT timeOfDay TO userOutput;
        ENDINSTANCE;
ENDMSC;
```

4.4.2 Analysis and Comparison with UCT

The UML notation uses Jacobsons's notation to describe use cases. This notation is informal and may contain many ambiguities since it is expressed in prose. The UML notation can help designers who need to have a high-level view of a system. No internal behaviour of the system is expressed in the use cases. The Jacobson's notation is supplemented by other design engines such as collaboration diagrams and sequence diagrams to express scenarios. However, all these notations are informal and there is no automatic way of converting from the scenario representations to the actual model development. Everything is done manually which can inevitably engender flaws.

Buhr's use case maps is a very helpful notation for expressing use cases. High-level designs, as well as low-level designs can be expressed using the notation. It has the power of showing behaviour patterns, scenarios, real-time behaviours and concurrency to name just a few. The visual appearance of the notation is very simple and easy to follow

when describing high-level designs. However one drawback is that the diagrams get really complicated when detailed use cases are described.

Right now there is no automatic way of jumping from use case maps to model development. There is ongoing work to provide this capability. A possible solution can be to generate use case trees, a formal notation, from use case maps and from there move on to the modeling phase. Use case maps are system-oriented and above message exchanges whereas use case trees ate component-oriented and based on message exchanges. Use case trees would be a refinement to use case maps. Also, use case maps are design-oriented whereas use case trees are more validation-oriented.

The MSC96 notation is very close to UCT. It has a well defined syntax which can be used to describe high level use cases, as well as detailed use cases. The possibility of grouping different MSCs under a single representation makes it very useful. However in both MSC96 and UCT, the scenarios have to be ungrouped before any model checking can be done. A model will run a single scenario at a time.

We think that UCT is clearer to write and understand than HMSCs. Consider the UCT example given below and the corresponding HMSC representation. S1 to S8 are SubTrees which are called by the MainTree.

```
Example                 MSC Example;
+S1                     EXPR L1;
^^+S2                       L1 : S1 SEQ (L2 ALT L3 ALT L4);
^^^^+S3                     L2 : S2 SEQ (L5);
^^+S4                       L3 : S4 SEQ (L9);
^^+S5                       L4 : S5 SEQ (L6 ALT L7);
^^^^+S6                     L5 : S3 SEQ (L9);
^^^^^^+S7                   L6 : S6 SEQ (L8);
^^^^+S8                     L7 : S8 SEQ (L9);
%%                          L8 : S7 SEQ (L9);
                            L9 : END;
                        ENDMSC;
```

The first difference is that the sequence of calls is clearer in the UCT representation because of the indentation. Also, the alternatives are clearly visible. Secondly, because the HMSC representation uses the "goto" concept, the construction or modification of big use cases can be very difficult. Another drawback of the HMSC notation is the generation of the basic MSCs from them. The structure of the representation makes it hard to select which basic MSCs are to be generated. However, in UCT we do not have this problem since we are dealing with trees. The selection of any node will generate the required basic MSCs, as described in section 6.4.

In MSC96 there are some concepts that are not available in UCT such as parallel composition, coregions, and instance creation and destruction. In future work we are thinking of using the power of concurrent TTCN to implement concurrency in UCT.

4.4.3 Example Showing the Compression Ratio of UCT

In the example given below, the use case trees are written one besides the other for readability. The events are represented by alphabetical letters for simplicity.

Example

A	S1	S2	S3	S4	S5
^^+S1	B	C	D	+S5	E
^^^^+S2	^^+S2	^^+S4	^^+S4	^^+S5	^^F
^^+S3	^^^^+S3	+S5	^^^^+S4	%%	G
^^^^+S4	%%	%%	%%		%%
^^^^^^+S5					
%%					

This use case tree will generate 704 MSCs ! This result can be obtained by replacing each SubTree call by all the possible combinations, or paths, of the actual SubTree. If the resulting paths still contain calls to other SubTrees, the process is repeated until they are free of SubTree calls. It is obvious, by looking at such a result, that use case trees are very efficient.

4.4.4 Limitations

The UCT notation is simple to write but one has to very careful while associating the indentation to a particular behaviour line since a mistake here can have bad consequences later on. The indentation may get confusing as the use case grows, and that is why, in such circumstances, it is advisable to break the use case into SubTrees. The use of two "^" characters to represent one indentation helps to reduce errors and improve readability.

Although use case trees offer the possibility of grouping many scenarios in a single representation, a real help to test designers, these scenarios still have to be ungrouped for the validation process. This is an extra step which cannot be avoided, at least for now, until a solution is found.

Another limitation of the notation is that it can only be used to describe a single actor at a time. But this can be an advantage if we want to test a single actor. We do not have to know what is inside it, the only concern being whether it is behaving as expected or not. Unlike MSCs which allow us to represent flow of messages between actors, use case trees restrict the flow of messages to the AUT only.

In this chapter we have presented the Use Case Tree notation. We have shown how several scenarios can be compacted into a single representation and how easy it is to do that. We have also shown how simple and clear the notation is and compared it with existing notations. The case study of applicability of UCT to ROOM and the fact that the semantics of UCT are based upon the practical international testing language TTCN provide evidence that UCT definitely scales up to large system development.

In the next chapter we will show how the Use Case Tree notation can be used in the use case driven validation framework that we are proposing.

Chapter 5

The Proposed Use Case Driven Validation Framework

In this chapter we propose a use case driven validation framework for software systems which are developed using an incremental process. Use cases have never been used before for validation activities and our proposition is meant to demonstrate that this is possible. Throughout this chapter we will explain, step by step, the different phases in the validation process. The validation process is an incremental iterative process since we are dealing with systems which are developed incrementally. At the end of the chapter we will do as assessment of the incremental iterative validation strategy.

5.1 Overview of the Proposed Framework

The proposed validation framework is suitable for systems which are built using an incremental development process. In this thesis we will consider the ROOM incremental development process. At each stage, a model is obtained, which can be run and tested. It is much better to test at an early stage because errors are easier to find in a less detailed model. If there are no errors after validation, the model can be further refined, otherwise, depending on the errors, the required modifications can be done.

The modeling phase and the pre-validation phase can be done in parallel. The pre-validation phase involves all the steps needed to prepare the use cases before the actual validation is done. Being able to model the system and prepare the use cases at the same time saves a lot of time. It is advisable that the use cases are not written by the person who is modeling the system. This is because if this person has misunderstood the behaviour of the system, he will do the same mistake, both in the model and the use cases.

Figure 5.1 shows the structure of the proposed validation framework. Artifacts are represented by round-edged boxes, whereas processes are represented by plain boxes. The normal flow of the process is shown by full arrows, whereas possible modifications are shown by dotted arrows. We may have to make corrections if ever a use case does not match the behaviour of a model. To better understand this, let us see broadly, what is happening in the framework. Later we will give a detailed description of each step.

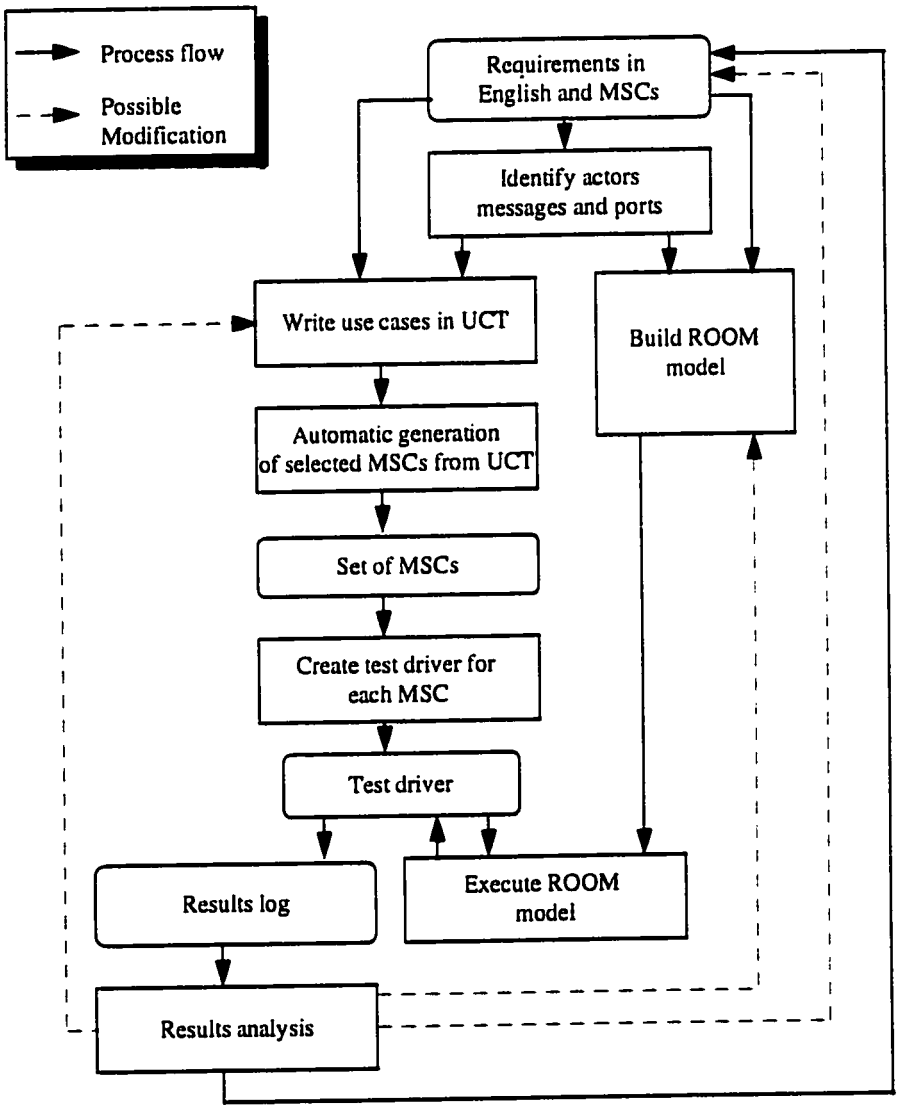


Figure 5.1

Figure 5.1 is to be read from top to bottom, starting with requirements. From requirements, actors, messages and ports are selected and these, along with the description in the requirements, will help to build the first model and write the associated use cases. As said earlier, these two processes can be done in parallel, and is shown as such in the figure. The pre-validation process involves writing the use cases in UCT, automatically translating them into MSCs, and selecting which MSCs will be used for validating the model. At this stage the model should be ready to be validated. The selected MSCs are then used to drive the model with the help of a test driver.

The main idea behind this validation framework is that the model (more precisely, the actor under test) should behave exactly as the MSC which is used to drive it since both of them are derived from the same requirements. If there are any discrepancies, there is either a problem in the model, the requirements, or the use cases and the appropriate modifications should be made, otherwise, the model can be further enhanced and more precise use cases written to continue the process. Now let us have a look at the steps involved, in more detail.

5.2 Steps for the Incremental Iterative Validation Strategy

5.2.1 Capturing Requirements

The first step in building a system is to capture the requirements. The primary function of requirements is to describe the system, giving the maximum information possible about its structure and behaviour. They are usually written in ordinary language or described using MSCs. But most of the time requirements are incomplete and ambiguous. It is very difficult to completely describe a system before it is actually built. Sometimes the model will contain more information than the requirements. For example, if some actors are to communicate in a system, a modeler may find it more appropriate to include an extra actor which will act as a coordinator. This will reduce the complexity of the model, by reducing the number of bindings in between actors. This new actor is an addition to the requirements, but does not change the behaviour of the system.

Ambiguity is common in requirements which are described using ordinary language. Sometimes the same actor may be referenced by two synonymous names. The

same thing may happen to messages. Writing requirements in prose is the easiest thing to do, but it has to be done with great care, since an ambiguous sentence may mean different things to different people. They have to be short and concise.

Sometimes MSCs are used to represent requirements for the system. They usually describe particular scenarios of the system's behaviour. They are more straightforward to understand than reading through a series of sentences, and, most important of all, they readily show the actors and messages which are involved.

Requirements, whether they are textual or graphical, are grouped together in a document which will be used in the next phase of the process, that is, identifying the actors, messages and ports.

5.2.2 Identifying Actors, Messages and Ports

Since the modeling phase and the use case writing phase are to be done by two different persons, it is very important that the same actor, message and port names be used by both, to avoid conflict in names later. To prevent this problem from happening, someone has to go through the requirements to identify the actors, the messages and the ports before the modeling starts or the use cases are written. Like capturing requirements, this is a manual process. The general rule is that, for requirements written in ordinary language, nouns are most of the time potential candidates for actors, whereas action verbs, which are done by or to a potential actor, are candidates for messages. Then, depending on the classification of the messages, protocol classes can be deduced, which will give the name of the ports.

Let us go through the fax system requirements, as described below, to show how we can derive the actors, the messages and the ports. The potential actors are underlined, whereas the potential messages are italicized. Repeated potential actors and messages having the same names should be marked only once for simplicity and readability.

The informal requirements, given in English language, is as follows :

"We want to model a fax machine.

To send a document, a user performs the following actions on the fax machine:

He *inserts the document* in the fax machine

He *dials the destination number* on a keypad

He *presses the go button*

The fax machine will then send *a connection request* to the destination fax machine. If the destination fax machine is free, it will *send a ring signal followed by an answer* to the sending fax machine. When the sending fax machine receives the answer, it will *transmit the data* to the destination fax machine. Then it will *show the time of the day*. In case the destination fax machine is busy, it will *send a busy tone* to the sending fax machine. The sending fax machine will then *display an error message* for five seconds followed by the time of the day."

The potential actors are :

- fax machine
- document
- user
- keypad
- go button
- destination fax machine
- sending fax machine

From this list we see that fax machine and sending fax machine refers to the same actor. Since all the description is done around the sending fax machine we prefer to choose the name sending fax machine rather than fax machine. So we should remove fax machine from the list.

Since a document is an entity which has no message interactions with any other actors, it will not be considered as an actor, and should be removed from the list. User and destination fax machine, although they do not form part of the sending fax machine, should be left because they interact directly with it, through exchange of messages.

The actors which are left after this selection will all form part of the system. For each of them a single-worded name has to be chosen.

The potential messages are :

- insert a document (paper)
- dial number (number)
- press go button (go)
- send ring signal (ring)
- send answer (answer)
- transmit data (data)
- show time of day (timeOfday)
- display error (error)
- request a connection (conReq)
- send busy tone (busy)

From this list we write one-worded names for the messages as indicated in parentheses. In our list, all messages are distinct and so no further modifications should be done.

The last step is to derive the port names. For this we have to first, group the messages, according to their functions, into protocol classes. In our example we can group the messages under three protocol classes, namely UserInput (paper, number and go), UserOutput (error and timeOfDay) and CommSig (conReq, ring, answer, data and busy). We can then choose port names, derived from the protocol classes.

With our list of actors, messages and ports, we are now ready to either write use cases or build a model for the system.

5.2.3 Writing Use Cases

Use case writing is an important process in the overall validation framework. It has to be done with extreme care because a mistake here may have bad consequences on the whole system. For example, if a model does not conform to a use case, most of the time we think that it is the model which is not well built, and start looking for modeling flaws. But, what if the mistake was done in the use case ? This is why use cases have to be reviewed with great attention and, if possible, by another tester.

When writing use cases, the objective is to find the minimum number of use cases to fully describe the system. But the problem here is, which ones to write ? If there is a finite set of requirements, use cases have to be written such that each one of the requirements is described. This will help to check whether a model is complete or not after the validation process. However, if requirements are constantly being modified, added or removed, then use case writing becomes more difficult.

In the proposed validation framework, use cases are written in UCT. Writing use cases in UCT is easier than writing them in another notation. There is a definite syntax that has to be followed, which reduces a lot of ambiguities. Also, a whole use case can be represented in a single use case tree, which promotes simplicity and clarity. However, it is very important to analyse the informal description of the requirements, especially when they are written in prose, to fully understand the behaviour of the system and write valid use cases.

Requirements which are described using MSCs are easier to convert into use cases because the entities, and required flow of messages, are already defined in them. Since a UCT can represent several MSCs, it is advantageous to be able to group related requirement MSCs together. In this way we obtain a compact group of MSCs, from which we can choose the desired scenarios, for the validation process. Right now this process can be done manually but in the future we are thinking of automating it.

5.2.4 Generating Selected MSCs from Use Cases

After the use cases have been written in UCT we can automatically generate selected MSCs from them. We have developed a tool for this purpose. This was possible because UCT has a well defined grammar. After developing a notation to represent use cases, the next step in the work was to automatically generate the MSCs we want from them. This is an important step in the validation framework for many reasons.

These generated MSCs will be used later to drive the model under construction. It is easier to write use cases in UCT, and then derive MSCs from them, rather than writing MSCs directly. A use case tree may contain several MSCs, and finding an automatic way to generate them is an added asset since there is a tremendous gain in time. A selective generation is very important so that all the MSCs are not generated if we want to use only a subset of the MSCs contained in a UCT. Selective generation of MSCs is time and cost efficient since it allows particular functionalities to be tested rapidly.

Since the generation is automatic, there is more reliability. Writing MSCs manually is a long process and the chance of making mistakes is very high. This is very

important because if there is a mistake in an MSC which is used to drive a model, and the latter passes the test, there may be a flaw in the model which will go undetected.

Another reason is due to the graphical representation of MSCs. Because UCT is a new textual notation, some people may prefer to see the use cases in the more common MSC graphical form, for familiarity reasons.

5.2.5 Driving the Model

The next step in the validation framework is to execute the already built model of the system, using the MSCs obtained previously. Note that each MSC will describe one particular behaviour of an actor (the AUT), showing the messages going in and out of the ports and in the required order. The idea here is to show that the actor which is under test will behave as described by the sequence of interactions of the MSC. To do that, the "IN" messages from the MSC are injected (this is a process in ROOM whereby a message is made to pass through a port) through the appropriate ports. If the model is correct, it should react to these "IN" messages by sending back the "OUT" messages, as defined in the MSC, through the appropriate ports.

```
MSC 1;
    INSTANCE AUT;
        IN a FROM A;
        OUT b TO B;
        IN c FROM C;
        OUT d to D;
        OUT e to E;
    ENDINSTANCE;
ENDMSC;
```

Figure 5.2

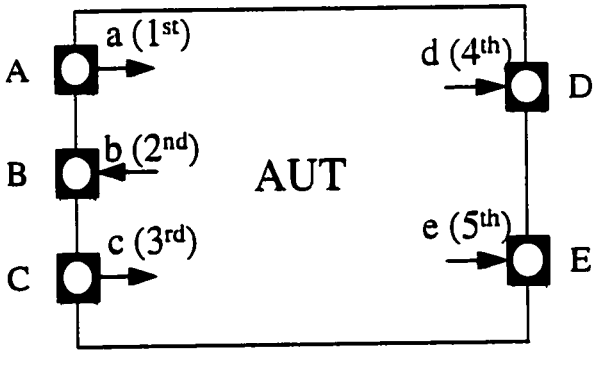


Figure 5.3

To make this clearer, let us consider a simple example. Figure 5.2 shows a MSC, obtained from a UCT, which describes the behaviour of an actor which has to be

validated. Figure 5.3 shows the structure of the AUT, in ROOM, and the order in which the messages are exchanged.

The AUT has five ports namely A, B, C, D, and E. To drive the actor, message "a" is first injected through port A. If the actor is behaving properly, it should send out message "b" on port B. When this message is received, message "c" is injected through port C. The actor should send out message "d" on port D followed by message "e" on port E. If the overall sequence of actions is successful, then the actor is behaving properly, otherwise there must be a flaw somewhere.

In this example, the injections were done manually (in the "running mode", ROOM allows the user to manually inject messages through the appropriate ports). But if we have to repeat the process for a large number of MSCs, manual injection becomes time consuming and unreliable. One alternative is to construct a "test driver" that will perform the appropriate message sends and in the proper order. For the purpose of this thesis, we will give a basic overview of how the test driver works. For more details please refer to [Selic96].

The test driver is basically, an actor in ROOM. For each MSC, a test driver is created automatically, and will be connected to the AUT. The sequence of actions in the MSC is represented as a ROOMchart and stored as the behaviour of the test driver. The test driver will control the flow of messages to and from the AUT. When one test is over, the results are stored in a log and the test driver is destroyed.

Let us consider the MSC of figure 5.2 as an example. The behaviour of the corresponding test driver is as shown in figure 5.4.

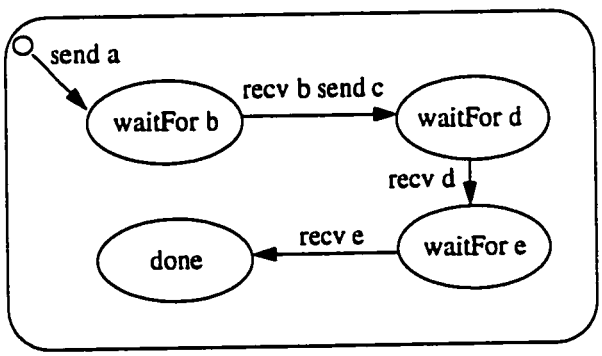


Figure 5.4

On the initial transition, message "a" will be automatically sent to the AUT through port A. The ROOMchart will then wait for message "b" to be received on port B, from the AUT. Then it will send message "c" on port C. The process will continue on until the state "done" is reached, indicating a pass, or a timeout occurs, indicating a failure. If the test driver is expecting a message from the AUT and does not get it after a certain amount of time, a timeout occurs to avoid indefinite waiting.

All the test results are stored in a result log. After the whole driving process is completed, the results can then be analyzed.

5.2.6 Results Analysis

The last step in the validation framework is to analyze the results obtained from the driving phase. In case an expected message was not received by a test driver, the MSC which was represented by this test driver would have been noted in the result log. Also, if the test driver had received some unexpected messages, these too would have been noted in the result log. Thus, the analysis of the result log can provide information on the errors which occurred during the validation process, showing where there are discrepancies between the requirements and the model. Usually the model and the requirements are checked first to see if the flaws are in one, or the other. If this is not the case then the use cases are checked.

Any changes to the model or the requirements may add new flaws to the model, and thus, more validation is needed. The validation process is iterated until there are no flaws in the model, hence the iterative aspect of the strategy. When a model is correct, further functionalities can be added and the whole process started again.

5.3 Assessment of the Incremental Iterative Validation Strategy

Incremental development is the process of constructing a partial implementation of a total system and slowly adding increased functionality or performance. When using incremental development, software is deliberately built to satisfy fewer requirements initially, but is constructed in such a way as to facilitate the incorporation of new requirements and thus achieve higher adaptability. Validation is an integrated part of the

incremental development process. Since the system is developed incrementally, so are the validation tests. For the same model, several tests can be repeated with minor changes to test particular functionalities. This is a case of iterative validation. The partially built system is continuously tested and, in this way, errors are found at an earlier stage which, in the long run, may be more economical.

Incremental iterative testing is very much used today. The "Spiral Model" for software development [Boehm 88] for example, incorporates an incremental iterative testing strategy into a well structured process where, after certain phases, an evaluation of the constructed system is done. Compared to the original "Waterfall Model", this is a much more efficient model, allowing more exchange between the developer and the customer through the analysis of prototypes. In case of disagreement, the appropriate decisions can be taken early in the development process and the necessary modifications made.

Most object-oriented techniques favour an incremental development process, thus allowing for incremental testing. In the Object Modeling Technique [Rumb91] for example, a system is developed by partially building an object model, a dynamic model and a functional model. If the overall model is satisfactory, further additions can be made to the three constituent models. In this way, a continuous link is kept between the three models which have to be relevant among each other. In case of inconsistencies, the appropriate modifications can be made.

These two previous examples show how beneficial incremental testing can be, when it is done in parallel with incremental development. The validation framework we proposed incorporate the incremental development process of ROOM with an incremental iterative validation strategy. At an early stage of development we can validate the model to see if it conforms to the requirements. We can imagine how inappropriate it can be to completely build a system and then validate it. The effects would be catastrophic if there are discrepancies. Such a strategy is beneficial in many ways.

First of all we have a big gain in time in the overall construction of a system. This is so because there can be two teams working in parallel, one building the model and one writing the use cases. When both are completed, the system can be validated, following

which it can be further developed. Secondly, in ROOM we can build a high level view of a system at first, which contain basic functionalities. The use cases are then written according to these functionalities, and are very simple. With further refinement of the model, more details can be added to the use cases. This gives an idea, right from the beginning, whether the right system is being developed. Also, the parties involved can discuss of any issues arising from incorrect requirements, right from the beginning of the development and thus reducing unsatisfaction later on. Our strategy is well structured and the steps are well defined. Apart from the use case writing, all the other processes are automated, which adds to time efficiency and reduction in errors. If all the requirements are specified in the use cases, then the completeness of a model can be determined; if all the MSCs generated from the use cases pass the validation process we may conclude that the model is complete.

In the process, a single actor is considered at a time. However, because ROOM is based on a hierarchical actor structure, the validation process can be scaled up to the highest level container actor. If we know that a container actor is behaving properly, we may assume that the internal actors are behaving properly too.

On the basis of all the benefits that we can get from an incremental iterative testing strategy and its wide use in other software development methodologies, we believe that our strategy is justified and can be readily applied to an incremental software development process.

In this chapter we have proposed a use case driven validation framework for software systems which are developed incrementally. This is an industrial strength process and the use of ROOM, a well known industrial methodology, as an example, is further proof of its potential. Furthermore, this strategy can be applied to other development methodologies following some modifications. For example the test driver can be modified to use the generated MSCs to either drive a model represented in the new methodology or be compared in another way to its execution. The choice of actors, messages and ports can be modified to suit the corresponding notation. We talked about the generation of MSCs from UCTs. In the following chapter we will present the tool automating it.

Chapter 6

User-Guided Automated Generation of MSCs from UCTs

In this chapter the tool for automatically generating selected MSCs from a UCT is presented. All the steps involved in the generating process are fully described and several examples are given to clarify the process. Most of the algorithms used in the process are elaboration of some common tree algorithms [Wirth76].

This chapter is the analysis and design behind the code presented in Appendix B. The tool has been optimized to improve performance and reduce memory use. The tool is also user-friendly. The tool has been thoroughly tested and, to our knowledge, works perfectly.

6.1 Overview of Tool

A tool for automatically generating MSCs in the Z.120 format, from use cases written in UCT is available. The tool is a syntax translator, meaning that if a use case written in UCT is inputted, the translator will convert each UCT construct into its equivalent MSC representation, without associating any meaning to the construct. For this reason, constructs which involve the evaluation of qualifiers or operations have not yet been implemented. One example is the Repeat construct, which involves evaluating a qualifier to decide whether further looping is required. The tool, as it is, is capable, from a use case written in UCT, to derive all or a subset of the MSCs involved, and describe for each, the communication events and the timer events, along with the risk associated to each scenario.

The tool will generate the chosen MSCs of a use case, in the textual format and will number them in order. If ever a risk is associated with a particular scenario in a use case tree, it will be written on the first line of the corresponding MSC. Figure 6.1 shows an example of a generated MSC.

```
MSC 1 (Normal Scenario);
        INSTANCE AUT;
                IN mess1 FROM A;
                IN mess2 FROM B;
                OUT mess3 TO A;
                IN mess4 FROM A;
                OUT mess5 TO B;
        ENDINSTANCE;
ENDMSC;
```

Figure 6.1

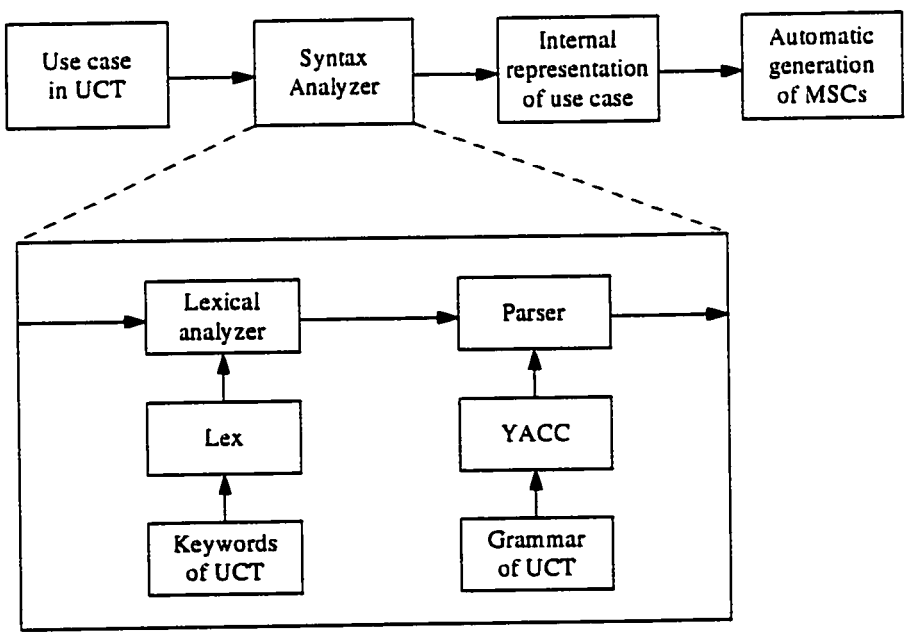


Figure 6.2

The tool is written in the C language and works on the UNIX and DOS operating systems. YACC, a grammatical analyzer, and Lex, a lexical analyzer, were used to build it. The structure of the tool is shown in Figure 6.2. There are three main parts in the tool, reading and analyzing the syntax of the use case, representing it in an internal form and finally generating the MSCs from this internal representation. The parts involved will be described in detail in the coming sections.

6.2 The Syntax Analyzer

To use the tool we need to have access to a file which will contain the pre-written use case, in UCT. The use cases have to be written in a text format using any text editor and saved as text files. For now, all use case trees involved in a use case have to be written in the same file, and one after the other. In the future, we are thinking of the option of saving use case trees in different files to promote reuse of some use case trees, especially those which are called several times. When the use case has been written, the next step is to pass it through the syntax analyzer.

The syntax analyzer will check if the use case given to it conforms to the UCT grammar. The syntax analyzer uses YACC and Lex. YACC is a program that will convert a formal grammar description of UCT, written in a form recognized by YACC, into another program called a YACC parser, that will parse any use case written in UCT. Lex, on the other hand, will use the keywords in UCT to produce a lexical analyzer, that will be used by the YACC parser. A detailed explanation of YACC and Lex can be obtained from [YACC86] and [Lex86] respectively. Along with the lexical analyzer and the YACC parser, an error-reporting routine will report any errors in the syntax of the use case, giving the line number, as well as the type of error. Figure 6.3 shows an example of a use case which contains an error, and the resulting error message generated by the syntax analyzer.

The error is on line 3 where the indentation is incorrect. Actually there should have been two "^" instead of one. But the parser will not say what is missing, on the contrary, it will say what is not expected. In our example, it will say that identifier B is not expected because the parser is expected to read a "^" character instead.

```
UseCaseFig6.3
A?mess1
^B!mess2
^^^^A?mess3
%%
```

The resulting error message is :
"Read from UCT file at line 3 : identifier B not expected"

Figure 6.3

To start the syntax analyzer we use the command : $*parse "filename"* where filename is the name of the file which contains the use case. In case the name is not given or an incorrect name is given, the program will produce an error message, and will stop. The parsing of the use case takes place on a line by line basis, just like an interpreter. If there is an error on a particular line, the parsing stops and the following lines are not taken into account. The error should be corrected to allow further parsing.

One important function of the tool is to represent a use case in an internal form which will be used later to generate the MSCs. This process takes place as the use case is being parsed. This means that if a use case has been completely parsed, without any errors detected, the internal representation would already have been constructed. In case there are syntax errors in the use case, the internal representation already built, is destroyed automatically. This parallel construction of the internal representation reduces the complexity of the tool.

6.3 Internal Representation of a Use Case

A use case is made up of use case trees which are basically, tree structures. Use case trees possess the two most important attributes of a tree, that is, nodes and branches. The starting node will be the root and the end nodes, the leaves. In our tool, we will associate each behaviour line of a use case tree to a node in a tree structure. This is why all events should be placed on a single line in UCT. The indentation of the events will help to construct the branches of the tree.

6.3.1 The Node

Each behaviour line of a use case written in UCT will be represented by a tree node in our tool. The node is a structure which contains all the information associated with a behaviour line, along with a link to the previous node and possibly a link to the right and left nodes, if they are available. The structure is as shown in Figure 6.4.

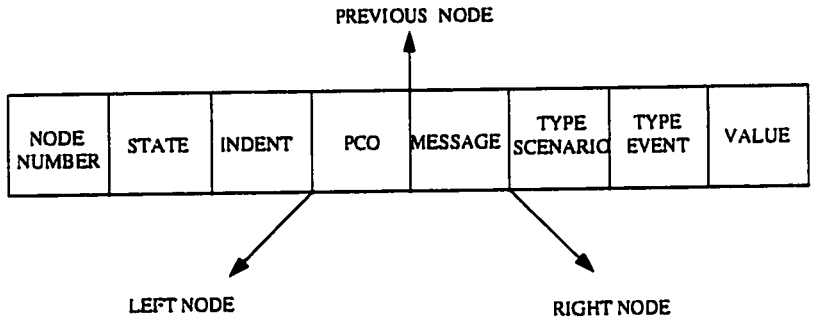


Figure 6.4

The following is an explanation of each field in the node structure :

- NodeNumber is an integer field which is used to number the nodes
- State is a character field and is used as a token in the generating phase, to indicate whether a node has been visited or not. It can take the following characters :

  S indicates that the node has never been visited before

  Y indicates that the node has been visited before, but has to be visited again to access other branches

  X indicates that the node has been visited before and does not need to be visited again

- Indent is an integer field and is used to store the indentation of the behaviour line. It is always an even number. The minimum value is 0. Indent is used to build the internal representation of a use case.

- PCO is a string field and is primarily used to keep the name of the port which is involved on the behaviour line. But, since there is no ports on a behaviour line which describes an Attach construct or a timer event, and since there is no timer event on a behaviour line which describes an Attach construct, the field PCO can also be used to store the TreeIdentifier and the TimerIdentifier, thus reducing the complexity and saving some memory space.
- Message is a string field and is used to store the name of the message involved in the communication event.
- TypeScenario is a character field and is used to keep the type of scenario associated with a node. Most of the time, only leaf nodes will use this field. The following characters are used for the different types of scenarios :

  N - indicates a normal scenario
  H - indicates a high risk scenario
  L - indicates a low risk scenario

- TypeEvent is an integer field. It is used to differentiate between the statements used in UCT and to know which one is contained in the node. The following values are used to represent each one of them.

1 -	Receive event	5 -	Cancel event
2 -	Send event	6 -	Otherwise event
3 -	Timeout event	7 -	Attach construct
4 -	Start event		

- Value is an integer field and is used to store the value associated with a timer.

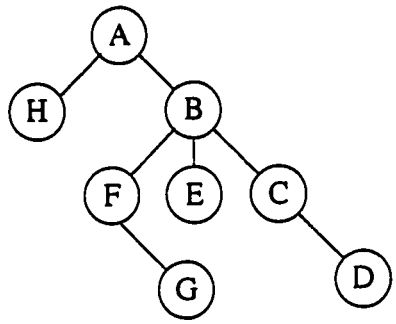
Nodes are connected to each other through pointers. Every node has to have a previous node, except the root node. A node may have only two branches, connected to a right node and a left node respectively. In the next section we will explain why this is so.

6.3.2 The Binary Tree

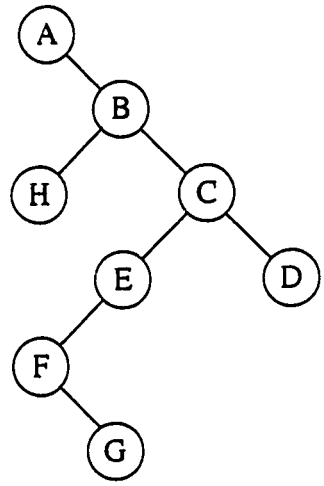
The number of branches in a use case tree is most of the time more than two, and this produces a non-binary tree, that is, a tree which may have more than two branches from a single node. From a computational point of view, it is easier to work with binary trees rather than complex structures representing non-binary trees. For this reason, all use case trees have to be represented as binary trees.

In our tool, we are not converting from a normal tree to a binary tree. We are just reading the use case tree as a binary tree. Figure 6.5 shows an example of a use case tree in a normal tree form and in a binary tree form.

```
UseCase
A!a
^^B!b
^^^^C?c
^^^^^^D!d
^^^^E?e
^^^^F!f
^^^^^^G?g
^^H!h
%%
```



Normal Tree Form



Binary Tree Form

Figure 6.5

The binary tree is constructed according to three basic rules :

1) If the indentation of the actual behaviour line is greater than the indentation of the previous behaviour line, then create a node on the right
2) If the indentation of the actual behaviour line is equal to the indentation of the previous behaviour line, then create a node on the left

3) If the indentation of the actual behaviour line is less than the indentation of the previous behaviour line, then go back until the first behaviour line with the same indentation is reached, and then create a node on the left

From Figure 6.5, we see that the four possible paths are

A B C D, A B F G, A B E, A H

Note that in the binary representation, a leaf is a node which does not have a right node. The algorithm used to read all the paths in the binary tree is described informally below. For the detailed algorithm please refer to Appendix B in the module "create.c".

0) The current node is the root.

1) From the current node, go down to the right, printing all the nodes which are not marked Y, until a leaf or a node marked Y is reached.

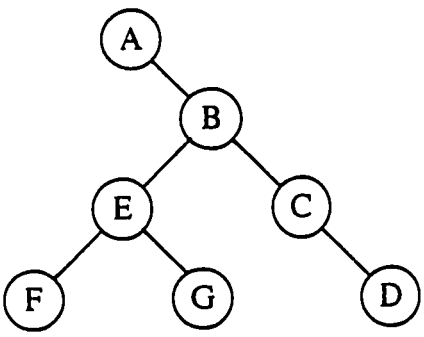
2) If a leaf which does not have a left node is reached, mark it as X, then go back, marking all the nodes which do not have a left node or which have a left node which is marked Y, with a X, until a node with a left node which is not marked Y is reached. Mark this node with a Y then go directly to the root and restart step 1. If a leaf which has a left node is reached, do step 3. If a node marked Y is reached, do step 4.

3) If a leaf which has a left node is reached, mark the leaf as Y, then go directly to the root and restart step 1.

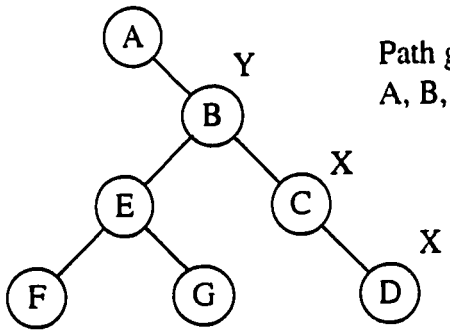
4) If a node marked Y is reached, go to the left until a node which is not marked Y is reached then restart step 1.

5) The reading algorithm is finished when the root is marked Y. Note that the root will eventually be reached while in the going back process, as described in step 2.

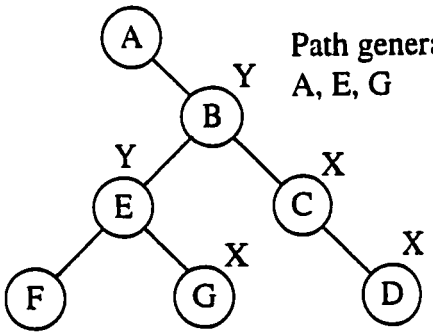
An example is given to better understand the algorithm. For the given binary tree there are three reading phases. Each phase shows the path read, and the state characters(X or Y) added on the way back, according to the algorithm described earlier. Note that the algorithm stops when the root is marked Y.



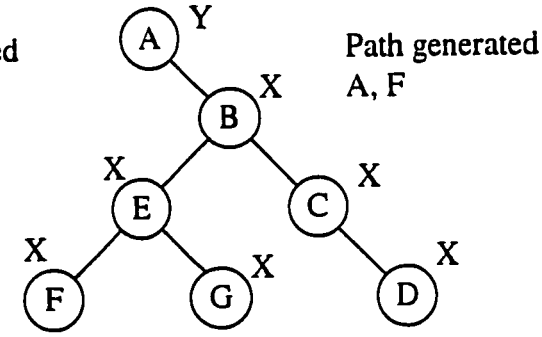
Initial Binary Tree



First reading



Second reading



Final reading

6.3.3 The Internal Use Case Structure

A use case is made up of a use case tree (MainTree) and, possibly, of other use case trees (SubTrees). These trees are read as binary trees when the use case is parsed. The MainTree is parsed first, followed by the SubTrees, in the order they appear in the file. But we need a way of storing these binary trees inside the computer for future

computations. The easiest way is to link all the binary trees through a linked list as shown in Figure 6.6.

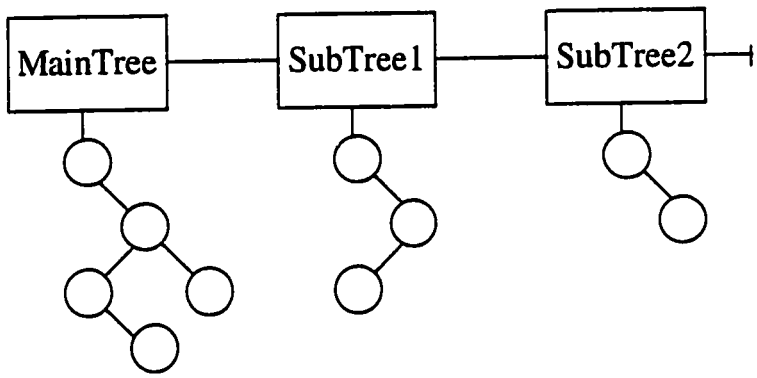


Figure 6.6

The MainTree binary tree is at the beginning of the linked list, followed by the binary trees of the SubTrees. With this kind of structure, we can access each line of the use case, just by going through the nodes of the different trees, since all the information of a behaviour line is contained in the nodes.

A MainTree may contain a lot of calls to SubTrees and so can SubTrees. The same SubTree can be called more than once. If every time we come across a SubTree call in a MainTree, we have to go and read all the possible paths of this SubTree, this will mean repeating the same process again and again. The best thing to do is to read the binary trees only once and store the paths in some kind of structure. The most appropriate structure for this kind of information storage is a double linked list. The double linked list is attached to the already built, link of binary trees as shown in Figure 6.7. Figure 6.7 is the internal representation of the use case described below it. Note that the nodes are given port names or SubTree names for simplicity.

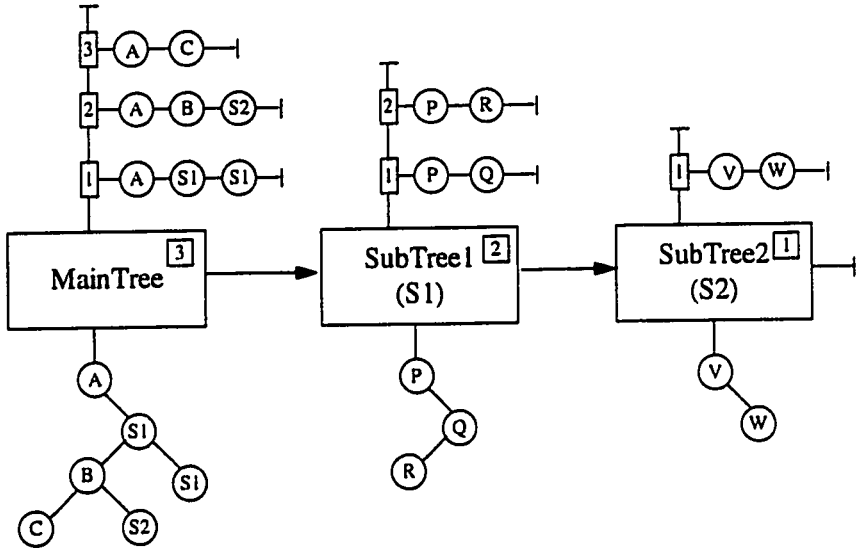


Figure 6.7

UseCase	S1	S2
A!a	P?p	V!v
^^+S1	^^Q!q	^^W?w
^^^^+S1	^^R!r	%%
^^B?b	%%	
^^^^+S2		
^^C!c		
%%		

The different paths, for each tree, are found using the binary tree reading algorithm, and stored with the corresponding tree, using a double linked list. The double linked list is basically a list of possible paths, each pointing to the corresponding list of nodes in the path. The nodes are inserted in the order they appear on the behaviour line. The list of paths is numbered, starting from one, for future computations. Also the number of paths for each tree is stored in a field in the main linked list.

With this internal representation of a use case, we have accomplished the first step for automatically generating MSCs. The generating process is presented next.

6.4 The Generating Process

Right now, we have a well defined internal use case structure, as seen in the previous section, from which we can start our generating process. But the only parts of the structure that we are going to use are the double linked list and the main list. We do not need to access the binary trees any longer because all the paths have already been stored in the double linked lists.

In case the MainTree does not have any calls to SubTrees, we do not have to do any more computations. Each list of nodes will give one MSC. But this is most of the time not the case. SubTrees are often used because of repetitive events, helping to structure the use case and reducing its size. We may sometimes find call to SubTrees following each other. Let us have a look at Figure 6.7 for example. The possible paths in the MainTree are : A S1 S1, A B S2, A C.

This is obtained by reading each list of nodes, from left to right. Now, from these paths, we have to get all the possible paths in the use case by replacing each call to a SubTree by the SubTree itself. Let us take the first path A S1 S1 of the MainTree for example. We have to replace each call to SubTree S1 by the paths P Q and P R as shown in Figure 6.8. We have to compute all the possible combinations of replacing each call to a SubTree by its constituent paths. In our example, S1 has two constituent paths, and thus there are two ways of replacing the first call to S1 and another two ways of replacing the second call. In all we get four possible combinations.

1	×	2	×	2	=	4 combinations
A		⌈P Q⌉		⌈P Q⌉		
		⌊P R⌋		⌊P R⌋		

Figure 6.8

The four combinations are A P Q P Q, A P Q P R, A P R P Q, A P R P R

A more difficult example would be a MainTree calling a SubTree, which in turn, calls another SubTree. Suppose one path of the MainTree is A S1 S2 and the paths in SubTree S1 are P Q and P R S2, and the paths in SubTree S2 are V W and V X. This is shown in Figure 6.9.

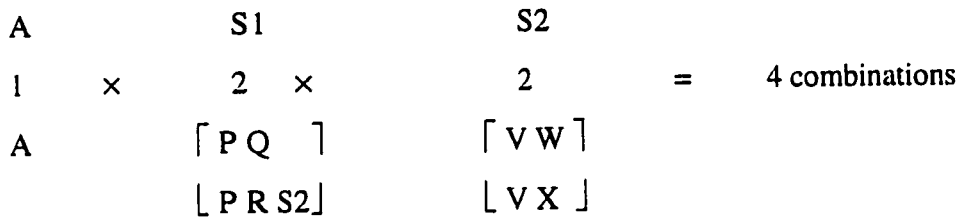


Figure 6.9

The four combinations are A P Q V W, A P Q V X, A P R S2 V W, A P R S2 V X

We see that two of the paths still contain calls to SubTree S2. So we have to do the process again and from A P R S2 V W we get A P R V W V W and A P R V X V W, and from A P R S2 V X we get A P R V W V X and A P R V X V X. And so the final combinations, or different MSCs, are :

A P Q V W A P Q V X A P R V W V W A P R V X V W
A P R V W V X A P R V X V X

Our tool uses the same technique to generate the MSCs. First of all, from a menu, we have the choice of generating all the MSCs or generate only those which contain a selected node. A node is selected by giving its node number as input in the program. In this way, MSCs corresponding to certain branches can be generated. A node can be chosen only at the MainTree level. An example is given in figure 6.10 which shows a UCT and the choice of possible MSCs which can be generated.

1 A

2^^B

3^^C

4^^^^D

5^^^^E

6^^^^^^F

Figure 6.10

If node 1 is selected all the MSCs will be generated since all paths contain event A. If node 2 is selected only path A, B is generated. If node 3 is selected paths A, C, D and A, C, E, F are generated. Selective generation of MSCs is possible because of the tree structure of UCTs.

The program will verify which paths in the MainTree contain the chosen node, and from there on, generate the MSCs. Each chosen path from the MainTree is traversed and, if it contains nodes which refer to SubTrees, these nodes are replaced by the appropriate list of nodes, obtained from the SubTree. In the process we are going to use a recursive approach because of the similarity of the steps involved, which are the same for the MainTree and the SubTrees.

The steps in the generating process are given below :

- Consider each chosen path of the MainTree in turn
- Follow one path to the end. If a node which refer to a SubTree is encountered, the number of paths associated with this SubTree, which is obtained by reading the value in the main linked list, is stored in an array called SubTreeArray. This is repeated if other calls to SubTrees are found, and finally, an array which contains the number of paths of all the SubTrees encountered on this MainTree path, is obtained.
- The next step is to calculate all the possible combinations of arranging the values of this array. This is analogous to calculating all the combinations in Figure 6.8. An example will help to clarify the step. Suppose the resulting array is [2, 1, 3]. We

know, from it, that the first SubTree on the path we read has 2 paths, the second SubTree we found has 1 path and the third SubTree has 3 paths. Note that the paths are represented as values in the array. So, in all, we have $2 \times 1 \times 3 = 6$ possible ways of writing the combination of SubTrees in the path, as shown below.

[1 1 1]
[1 1 2]
[1 1 3]
[2 1 1]
[2 1 2]
[2 1 3]

This is basically a 2×2 matrix which can be represented by a 2-dimensional array. This 2-dim array is computed from the array SubTreeArray. Each row of the array will correspond to one combination of the SubTrees, included in one path of the MainTree. Each value of the element of the 2-dim array refers to the path number in the corresponding SubTree.

- The next step is to create, for each combination in the 2×2 matrix, a new path which contains the paths of the SubTrees, similar to figure 6.9. Each new path which is created is checked to see if it does not contain any other calls to SubTrees. If it does then the whole process of reading though the path and creating an array SubTreeArray is done again for this particular path and hence the recursive nature of the algorithm. Otherwise, the path is printed out. Each node corresponds to a MSC construct and is dependent on the field TypeEvent. Figure 6.11 shows the possible values of TypeEvent and the corresponding MSC construct. After the new path is printed, it is deleted to avoid overuse of memory.

TypeEvent	MSC equivalent
1	IN <message> FROM <actor>;
2	OUT <message> TO <actor>;
3	timeout <TimerIdentifier>;
4	set <TimerIdentifier> <value>;
5	reset <TimerIdentifier>;

Figure 6.11

- The algorithm ends when all the possible paths have been printed out.

Let us now go through an example to show the whole process, stepwise. Consider Figure 6.7 which shows the internal structure of a use case. We assume that we only have to find all the possible paths of the first path of the MainTree.

- Choosing the first path in the MainTree, that is A S1 S1, we go through it to see if it contains any call to SubTrees. In our example it does. We update the first element of our array SubTreeArray with the number of possible paths in S1, which is 2. The second element in our array will also be 2, because we have another call to S1. Our array will contain only 2 elements because there are only two calls to SubTrees on this path and is shown here as [2, 2].
- The next step is to find all the possible combinations of our array and build a 2-dim array. The resulting 2-dim array is shown below :

[1 1]
[1 2]
[2 1]
[2 2]

Since the 2-dim array has four rows, we know that we will get four paths and hence we have to read the MainTree path A S1 S1 four times to build the respective paths. For each combination we have to create a new path, which will be destroyed after it has been printed out. On the first reading, we consider the first row in the 2-dim array. The first node, which is A, is inserted in the new path. On the first S1, we go and read path number 1 of S1, which corresponds to value 1 of the first element of the first row in the 2-dim array. The nodes P and Q are inserted in the new path, right after the node A. On the second S1, we go and read path number 1 of S1, which corresponds to value 1 of the second element of the first row in the 2-dim array. The nodes P and Q are inserted in the new path, right after the node Q. Since there is no more nodes to be added to the new path, it is checked to see if it contains any call to SubTrees. In our example it does not. So we print the MSC associated with this new path and delete it. Then we consider the second combination of the 2-dim array. First of all a new path is created. This time we will consider the first path of S1 for the first S1 which is read, and the second path of S1, that is P R, for the second S1 which is read because of the values [1 2] in the second row. When the reading is completed, the new path is printed and then deleted. This process is repeated for all the rows of the 2-dim array.

- The generation is complete when all MSCs have been printed out.

In this chapter the tool for automatically generating selected MSCs from a UCT has been presented. The most efficient algorithms have been used for the generating process. The next phase will be to use the tool in a real validation process. This is presented in the next chapter. The tool is used to generate MSCs from use cases written in UCT for the validation of a fax system.

Chapter 7

Case Study : Applying the Use Case Driven Validation Strategy to the Development of a Fax System

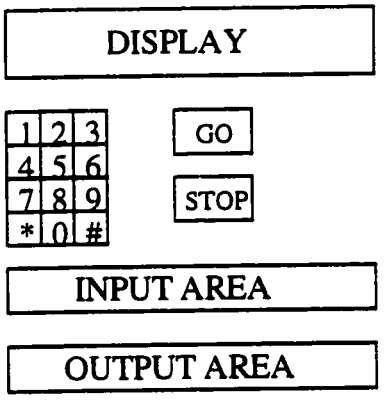
In this chapter we will present an example of a system and show how our validation strategy can be applied. We chose a fax system because it is fairly simple to understand, and can be easily developed using ROOM. Also, since we have been using the example of a fax system throughout this thesis, we think that it is more appropriate to stay with the same system, but, for this chapter, present a more detailed view of the system. We do not pretend that this fax system is complete in any ways or that it represents exactly the structure and behaviour of a normal fax system. This is only an example which is meant to present our work.

The main entity of this system is the FPS2000, which stands for "Fax Plain and Simple" [Prob94]. Our goal is to model this fax according to the requirements. But to do so, we also have to model the other entities to which this one is interacting, for validation purposes. If we model only the FPS2000, only the latter can be validated. The interactions between the FPS2000 and the other entities, for example the user or the destination fax machine, will not be. Since the FPS2000 is one entity of a whole system, everything in the system has to be modeled.

In this chapter we will describe the capturing of the requirements, the identification of actors, messages and ports, the building of the model in two increments, the writing of a few use cases in UCT for each increment, the generation of MSCs from the UCTs, and finally the validation process.

7.1 Capturing the Requirements of the Fax System

Figure 7.1 shows a simple view of the Fax Plain and Simple (FPS2000). It has an input area to insert documents, an output area to receive documents, a keypad to dial the destination phone number, a go button to send a fax, a stop button to cancel a transmission and finally, a display panel to display messages.



The display panel is capable of displaying :

"TransmissionOK"

"Time"

"Transmitting"

"Dialing"

"Receiving"

"ReceptionOK"

"Document Ready"

"Error"

Figure 7.1

The purpose of the FPS2000 is to send and receive facsimile documents.

Requirement 1 - Sending a document (normal behaviour)

- A user inserts a document into the input area. The display panel should display "Document Ready".
- He then enters the destination number, one digit at a time. The display panel should display the number.
- Finally he presses the GO button. The display panel should display "Dialing". The FPS2000 will send a connection request to the destination fax. A ringing tone will be heard and when an answer is received, the data will be sent. The display panel should display "Transmitting".

- If another page is inserted in a period of 10 seconds following the start of the transmission, this will also be transmitted and the display panel should display "Transmitting".
- If no page is inserted after 10 seconds, the FPS2000 should send a signal to the destination fax saying that the transmission is over. Then it should wait for a signal from the destination fax saying whether the transmission was OK.
- If the transmission is OK the FPS2000 should send a disconnection signal to the destination fax to break the connection.
- At the end of a successful transmission the display panel should display "TransmissionOK" followed by "Time".

Sending a document (exceptional behaviour)

- If the STOP button is pressed at any time, sending is aborted. The display panel should display "Error" for 5 seconds followed by "Time".
- If the FPS2000 does not get an answer from the destination fax after a period of 20 seconds, transmission should be aborted and the display panel should display "Error" for 5 seconds followed by "Time".
- If the FPS2000 sends a connection request to a destination fax and receives a busy signal, transmission should be aborted and the display panel should display "Error" for 5 seconds followed by "Time".
- If the FPS2000 is waiting for paper to be inserted or the number to be entered or the GO button to be pressed and receives a connection request from a fax machine it should stop the sending process and jump to a receiving state. It should send the ring signal to the calling fax machine followed by the answer signal.
- If the FPS2000 receives a connection request after the GO button is pressed, it should send a busy signal to the calling fax machine.
- If the FPS2000 is communicating with a destination fax and it receives a connection request from a third fax, it should send a busy signal to the third fax.

Requirement 2 - Receiving a document (normal behaviour)

- If the FPS2000 is in an idle state or has not yet established a connection with a destination fax machine, and it receives a connection request from a fax, it will send a ring signal followed by an answer signal to this calling fax, saying that it is free to receive.
- When it receives data, the display panel should display "Receiving". The received document should be outputted through the output area.
- If after the successful reception of a fax, the FPS2000 receives an end of transmission signal from the sending machine, it should send a signal in return saying that the transmission was OK.
- If it then receives a signal of disconnection, the display panel should display "ReceptionOK", followed by "Time".

Receiving a document (exceptional behaviour)

- While the FPS2000 is receiving a document, any input by the user will be discarded.
- If the FPS2000 receives a connection request from another fax machine it should send a busy signal.
- If the FPS2000 does not receive an end of transmission signal before a period of 15 seconds, the receiving should be canceled and the display panel should display "Error" for 5 seconds, followed by "Time".

7.2 Identification of Actors, Messages and Ports

The next step is to identify all the actors, messages and ports involved in the system before the model is built or use cases are written. We will only show a list of what was obtained without going through the whole process of identifying the verbs and nouns, as explained in section 5.2.2.

Actors

AFaxSystem	- represents the overall system
FPS2000	- the fax which is being modeled
User	- the person interacting with the FPS2000
DestinationFax	- the fax with which the FPS2000 will communicate
ThirdFax	- the fax which will interact with the FPS2000 when it is communicating with the DestinationFax
InputArea	- the area in the FPS2000 where paper will be inserted
OutputArea	- the area in the FPS2000 where paper will be received
Keypad	- the place in the FPS2000 where the number will be dialed
GoButton	- the button in the FPS2000 which will start the transmission
StopButton	- the button in the FPS2000 which will stop the sending process
Display	- the area in the FPS2000 where the information will be displayed
Controller	- this actor is not in the requirements but has been added to synchronize the message interactions inside the FPS2000. The behaviour of the FPS2000 is inside the Controller.

Messages

userInputPaper	- from User, indicating that a document has been inserted
userDialNumber	- from User, indicating that a digit has been entered
userPressGo	- from User, indicating that this button has been pressed
userPressStop	- from User, indicating that this button has been pressed
userGetPaper	- to User, indicating that the user has received the fax
userSeeDocReady	- to User, showing to the user that the document is ready
userSeeDialing	- to User, showing that dialing is taking place
userSeeTransmitting	- to User, showing that transmission is taking place
userSeeReceiving	- to User, showing that a document is being received
userSeeTransOK	- to User, showing that a document was successfully transmitted

userSeeReceptionOK	- to User, showing that a document was successfully received
userSeeError	- to User, showing that an error has occurred
userSeeTime	- to User, showing the current time
paperIn	- from InputArea, indicating that a paper has been inserted
documentReady	- from Controller, showing that the document is ready
paperOut	- from Controller, indicating that a fax has been received
digitIn	- from Keypad, indicating that a digit has been entered
digitDisplay	- from Controller, indicating which digit has been entered
goPressed	- from GoButton, indicating that it has been pressed
stopPressed	- from StopButton, indicating that it has been pressed
conReq	- from or to Controller, indicating a connection request
ring	- from or to Controller, indicating a ring tone
answer	- from or to Controller, indicating an answer
dialing	- from Controller, indicating that dialing is taking place
data	- from or to Controller, indicating data sent
transmitting	- from Controller, indicating transmission
transOK	- from or to Controller, indicating transmission successful
transmissionOK	- from Controller, indicating that a fax has been successfully transmitted
receiving	- from Controller, indicating fax being received
receptionOK	- from Controller, indicating reception successful
endTrans	- from or to Controller, indicating end of transmission
disconnect	- from or to Controller, indicating a disconnection
busy	- from or to Controller, indicating a busy signal
error	- from Controller, indicating an error
time	- from Controller, indicating current time

Ports

To find port names we first have to find the protocol classes, by grouping together related messages. The protocol classes of our system can be :

userInput	: userInputPaper, userDialNumber, userPressGo, userPressStop
userOutput	: userGetPaper, userSeeDocReady, userSeeDialing, userSeeTime, userSeeTransmitting, userSeeReceiving, userSeeTransOK, userSeeReceptionOK, userSeeError
input	: paperIn, digitIn, goPressed, stopPressed
output	: paperOut, documentReady, digitDisplay, dialing, error, time, transmitting, transmissionOK, receiving, receptionOK
commSig	: conReq, ring, answer, data, endTrans, transOK, disconnect, busy

The ports will correspond to the protocol classes. For clarity and simplicity, the port names are derived from the protocol class names.

7.3 Building the ROOM Model of the Fax System

This section will present the construction of the ROOM model of the fax machine in two increments. First we will build the initial model according to the first set of requirements, "Sending a document". We are only going to present the structure and behaviour models that provide the most information about the fax system. Then, as an example, two use cases for this initial model will be written using the UCT notation and all the MSCs will be generated from them using our tool. These MSCs will be a subset of all those to be used to validate the model.

After the initial model is validated the final model will be built by the addition of the second set of requirements "receiving a document", and the same validation process will be repeated. We will present only two use cases as examples and the MSCs generated from them.

For the validation process we will use the manual injection technique described in section 5.2.5. We will not use the "test driver" automated technique since the present version of the ObjecTime toolset does not support it yet.

7.3.1 Initial Model of the Fax System

The initial ROOM model of the fax system is based on the set of requirements for Sending a Fax. Two structure models are presented, one being the actor AFaxSystem which gives an overview of the system with the User, the FPS2000, and the DestinationFax, as shown in figure 7.2, and the other one the specific components of the FPS2000, as shown in figure 7.3. In figure 7.2, the actor FPS2000 is a container actor. Also note that the controller is not part of the requirements but has been added only to simplify the message interchange.

Figure 7.4 represents a high level view of the sending process. The state "sending" is a hierarchical state and figure 7.5 shows its internal ROOMchart. The transition "stop" is a group transition and can be triggered from any state inside the state "sending". The transition "default" in figure 7.4 is the transition which is taken whenever an unexpected message is received.

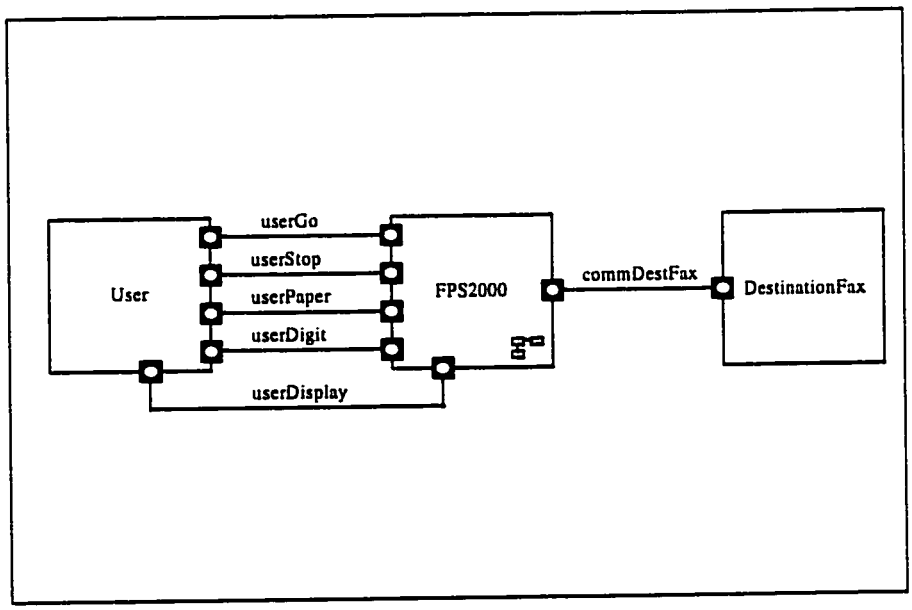


Figure 7.2 (AFaxSystem - Initial model)

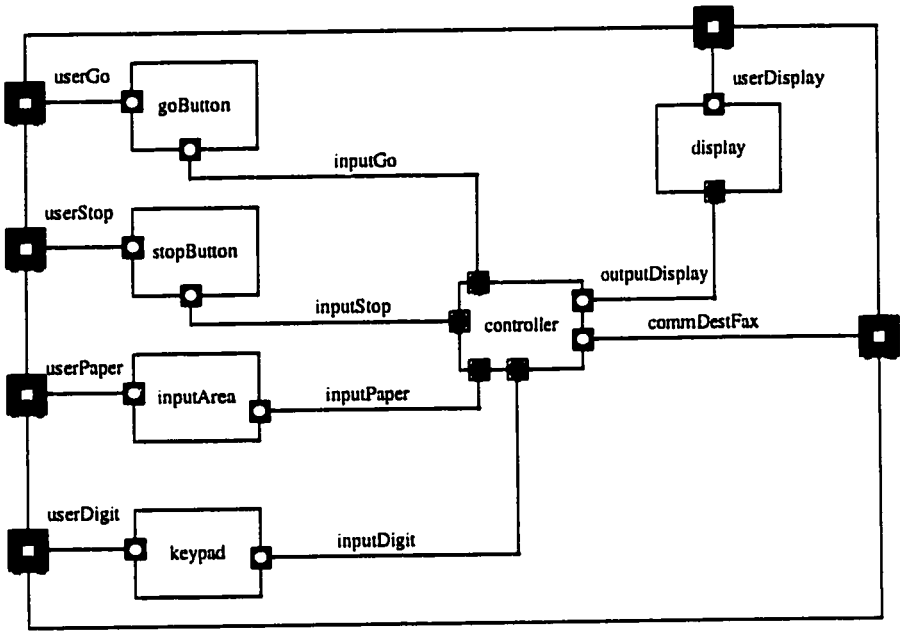


Figure 7.3 (actor FPS2000 - Initial model)

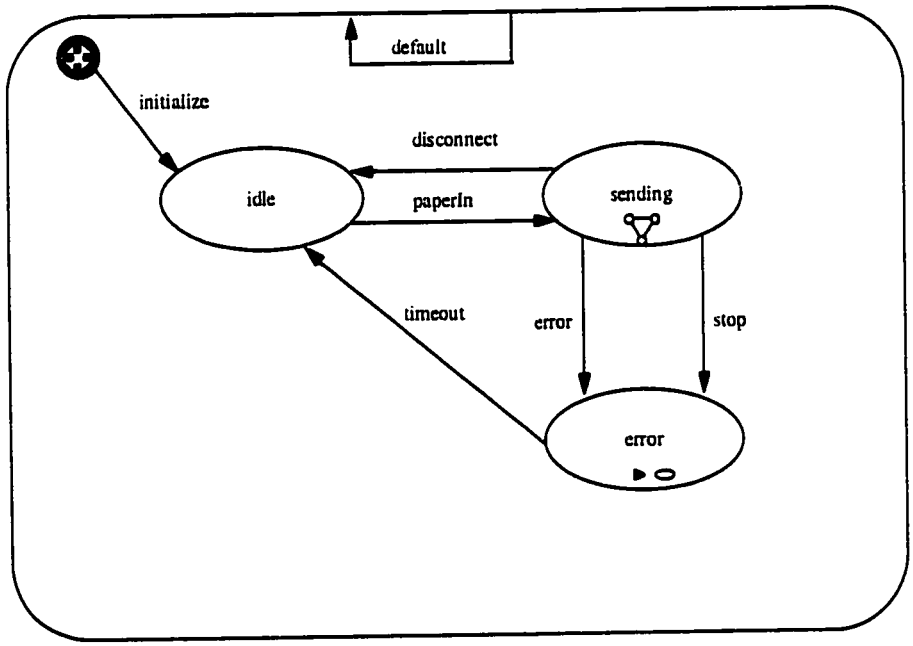


Figure 7.4 (behaviour of controller - Initial model)

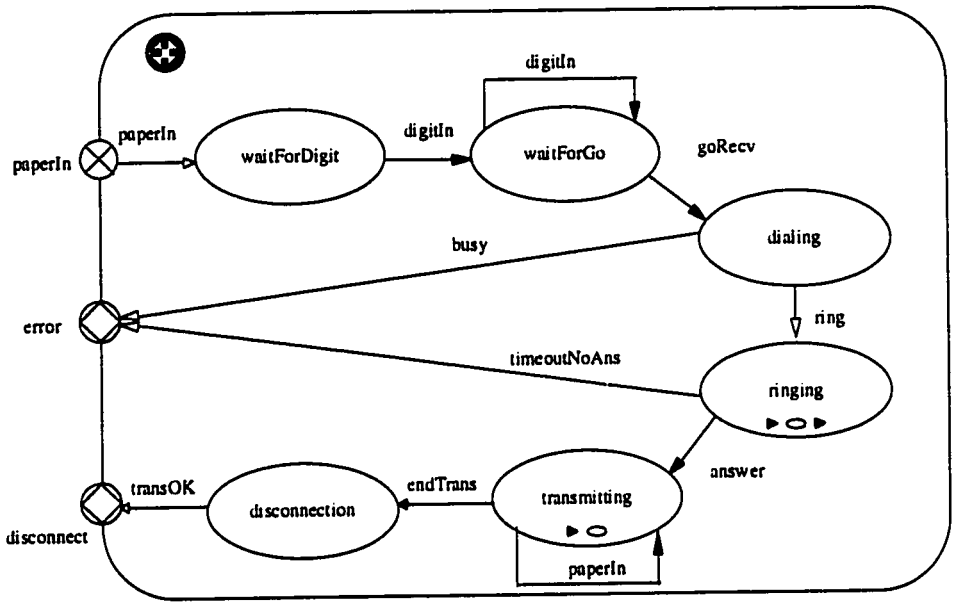


Figure 7.5 (inside sending state - Initial model)

7.3.2 Use Cases and Generated MSCs for the Initial Model

In this section we are going to present two examples of use cases written in UCT, and their corresponding MSCs, generated from our tool, intended for the validation of the initial model of the fax system. These use cases are by no means exhaustive, and are meant only to describe some of the message interactions in the fax system.

One obvious role of the FPS2000 is to send faxes. When a fax is sent, there is a well defined sequence of messages which go in and out of the FPS2000 , and these will be described in our use cases. The FPS2000 will be the actor under test. The first use case is "SendingAFax". For this use case we will consider the normal scenario and two exceptional scenarios. The next use case is the "FeudingFaxes". This is a use case which describes what happens when two fax machines try to send a fax to each other at the same time. Usually there is a race condition and the fastest will win. In our requirements however, we specified that whenever a connectionRequest is sent, any other connectionRequests which are received are discarded and a busy signal is sent back. This is expressed in the use case "FeudingFaxes". Note that this example is not exactly the same as in [Prob94], where the use case is described when the FPS2000 is in manual mode. We assume in our case study that the FPS2000 operates in automatic mode only.

UseCaseSendingAFax

+Preamble

^^+NormalTransmission N

^^+DestinationBusy H

^^+NoAnswerFromDestination H

%%

Preamble

userPaper?paperIn

^^userDisplay!documentReady

^^^^userDigit?digitIn

^^^^^^userDisplay!digitDisplay

^^^^^^^^userGo?goPressed

^^^^^^^^^^commDestFax!conReq

^^^^^^^^^^^^userDisplay!dialing

%%

NormalTransmission

commDestFax?ring

^^commDestFax?answer

^^^^commDestFax!data

^^^^^^userDisplay!transmitting

^^^^^^^^+Postamble

%%

DestinationBusy

commDestFax?busy

^^+Error

%%

NoAnswerFromDestination

commDestFax?ring

^^START T1(20)

^^^^?TIMEOUT T1

^^^^^^+Error

%%

For the purpose of helping the reader understand which parts of the use case trees correspond to the generated MSCs we will annotate the MSCs of this use case with the corresponding use case trees. This will be done for this example only.

Postamble

commDestFax!endTrans

^^commDestFax?transOK

^^^^commDestFax!disconnect

^^^^^^userDisplay!transmissionOK

^^^^^^^^userDisplay!time

%%

Error

userDisplay!error

^^START T2(5)

^^^^?TIMEOUT T2

^^^^^^userDisplay!time

%%

Generated MSCs for use case "SendingAFax"

MSC 1 (Normal Scenario);

INSTANCE ActorUnderTest;

IN paperIn FROM userPaper;	
OUT documentReady TO userDisplay;	
IN digitIn FROM userDigit;	
OUT digitDisplay TO userDisplay;	Preamble
IN goPressed FROM userGo;	
OUT conReq TO commDestFax;	
OUT dialing TO userDisplay;	
IN ring FROM commDestFax;	
IN answer FROM commDestFax;	NormalTransmission
OUT data TO commDestFax;	
OUT transmitting TO userDisplay;	
OUT endTrans TO commDestFax;	
IN transOK FROM commDestFax;	
OUT disconnect TO commDestFax;	Postamble
OUT transmissionOK TO userDisplay;	
OUT time TO userDisplay;	

```
        ENDINSTANCE;
ENDMSC;

MSC 2 ( High Risk Scenario );
        INSTANCE ActorUnderTest;
                IN paperIn FROM userPaper;                ┐
                OUT documentReady TO userDisplay;         |
                IN digitIn FROM userDigit;                |
                OUT digitDisplay TO userDisplay;          |   Preamble
                IN goPressed FROM userGo;                 |
                OUT conReq TO commDestFax;                |
                OUT dialing TO userDisplay;               ┘
                IN busy FROM commDestFax;                 |   DestinationBusy
                OUT error TO userDisplay;                 ┐
                OUT time TO userDisplay;                  ┘   Error
        ENDINSTANCE;
ENDMSC;

MSC 3 ( High Risk Scenario );
        INSTANCE ActorUnderTest;
                IN paperIn FROM userPaper;                ┐
                OUT documentReady TO userDisplay;         |
                IN digitIn FROM userDigit;                |
                OUT digitDisplay TO userDisplay;          |   Preamble
                IN goPressed FROM userGo;                 |
                OUT conReq TO commDestFax;                |
                OUT dialing TO userDisplay;               ┘
                IN ring FROM commDestFax;                 |   NoAnswerFromDestination
                OUT error TO userDisplay;                 ┐
                OUT time TO userDisplay;                  ┘   Error
                ENDINSTANCE;
ENDMSC;
```

UseCaseFeudingFaxes

+Preamble

^^commDestFax?conReq

^^^^commDestFax!busy

^^^^^^commDestFax?busy

^^^^^^^^+Error

%%

Preamble

userPaper?paperIn

^^userDisplay!documentReady

^^^^userDigit?digitIn

^^^^^^userDisplay!digitDisplay

^^^^^^^^userGo?goPressed

^^^^^^^^^^commDestFax!conReq

^^^^^^^^^^^^userDisplay!dialing

%%

Error

userDisplay!error

^^START T1(5)

^^^^?TIMEOUT T1

^^^^^^userDisplay!time

%%

Generated MSC for use case "FeudingFaxes"

MSC 1;

INSTANCE ActorUnderTest;

IN paperIn FROM userPaper;

OUT documentReady TO userDisplay;

IN digitIn FROM userDigit;

OUT digitDisplay TO userDisplay;

IN goPressed FROM userGo;

OUT conReq TO commDestFax;

```
            OUT dialing TO userDisplay;
            IN conReq FROM commDestFax;
            OUT busy TO commDestFax;
            IN busy FROM commDestFax;
            OUT error TO userDisplay;
            OUT time TO userDisplay;
        ENDINSTANCE;
ENDMSC;
```

7.3.3 Validation of the Initial Model

The validation of the initial model was done manually, using the same technique described in section 5.2.5. As the two examples given before, all the requirements of the sending process were expressed in UCTs and the generated MSCs used to drive the model. The results were satisfactory. After some minor modifications, the model behaved exactly as described by the MSCs. From the result we could conclude that the initial model was complete with respect to the sending process requirements.

The next step is to add more functionalities to the model and to do so we will consider the "receiving" requirements. This will be the last increment to produce the final model of the fax system.

7.3.4 Final Model of the Fax System

To obtain the final model of the fax system the initial model is modified according to the requirements for receiving a fax. All the previously presented structure and behaviour models are modified to reflect the added functionalities. The actor AFaxSystem is modified to include the actor ThirdFax as shown in figure 7.6. This actor is used to interact with the FPS2000 when the latter is communicating with the DestinationFax. For example we can check whether the FPS2000 behaves correctly when it receives a connectionRequest while it is communicating with the DestinationFax. An extra ROOMchart is added to represent the receiving process as shown in figure 7.10. The final model is given below.

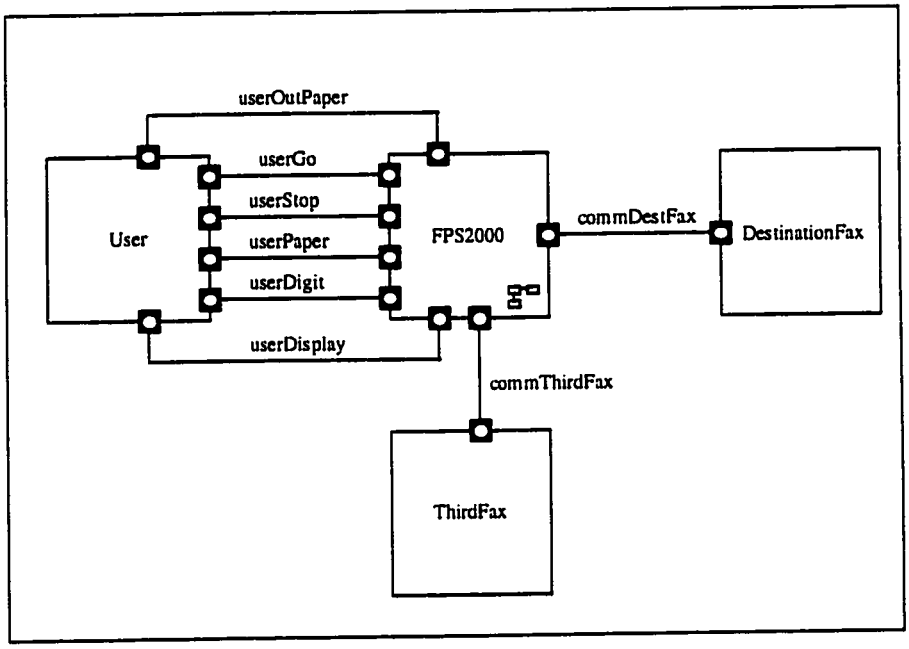


Figure 7.6 (AFaxSystem - Final model)

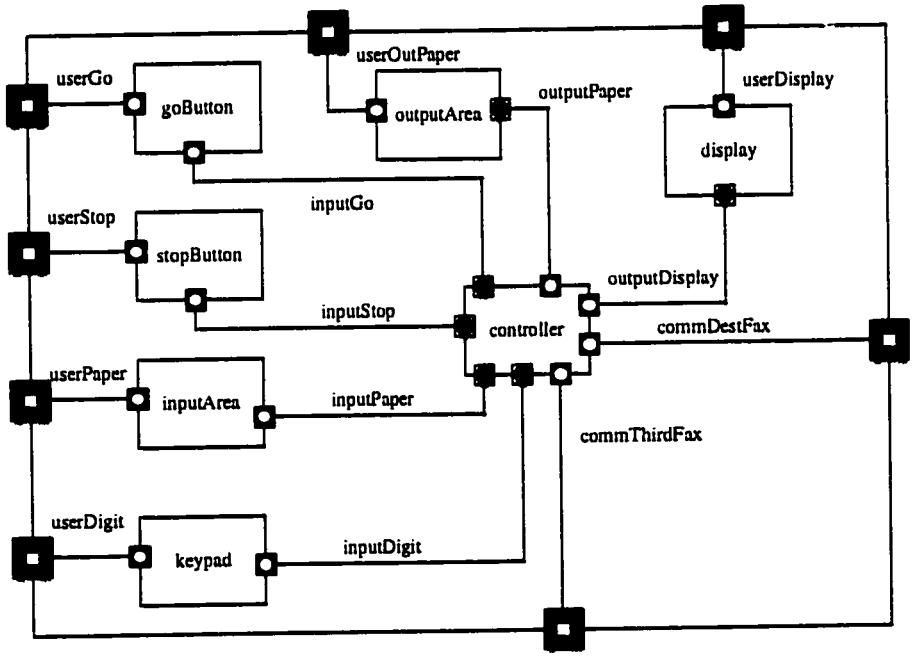


Figure 7.7 (actor FPS2000 - Final model)

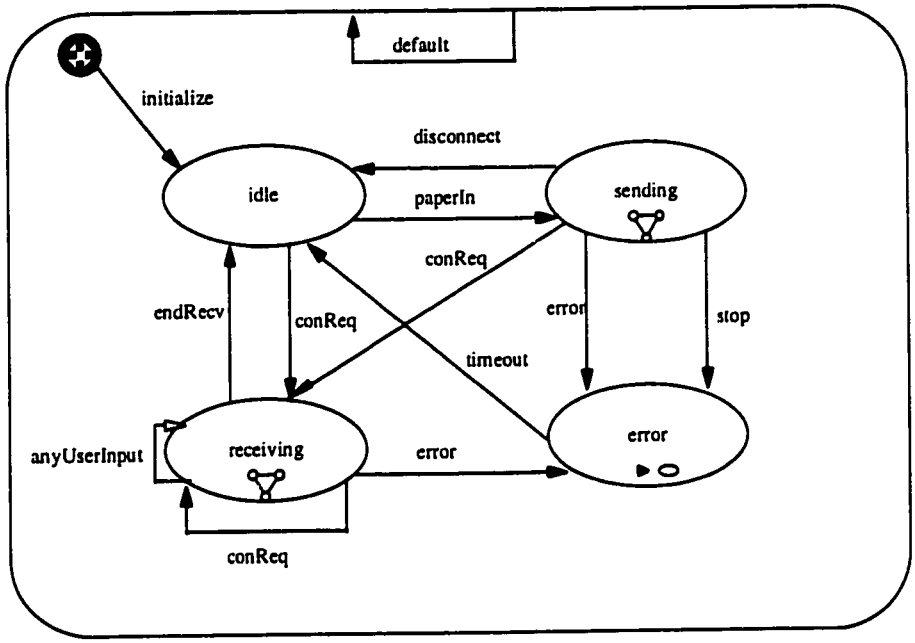


Figure 7.8 (behaviour of controller - Final model)

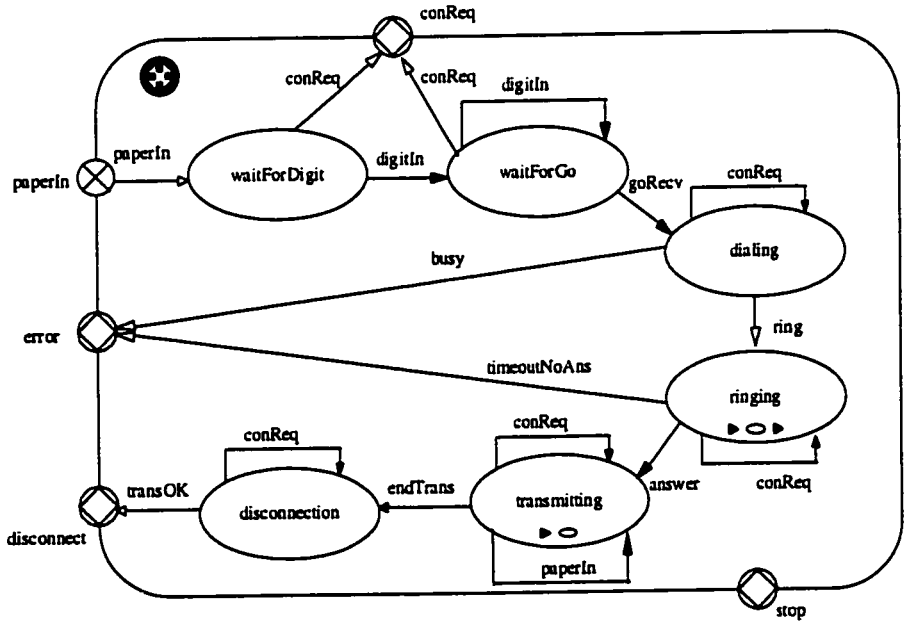


Figure 7.9 (inside Sending state - Final model)

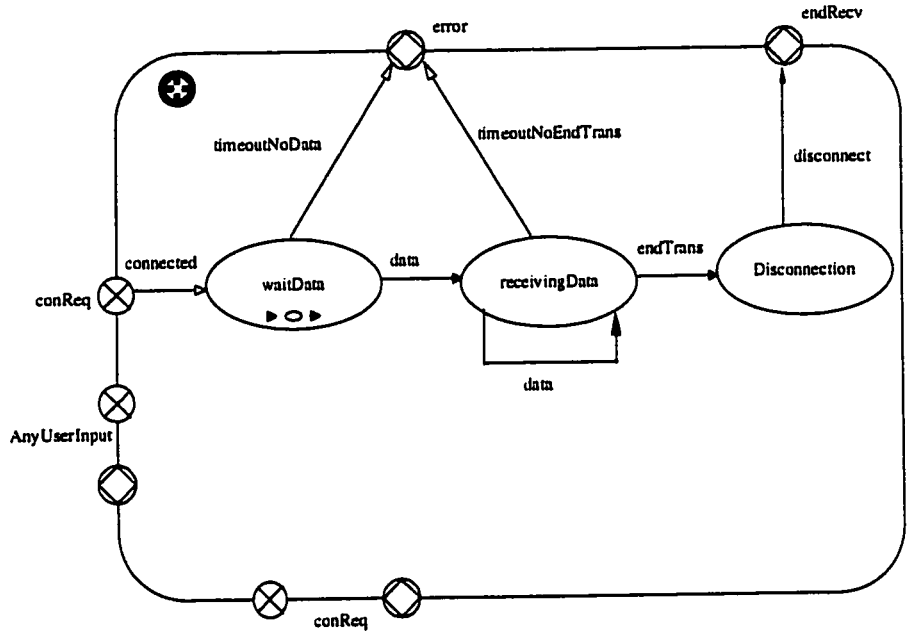


Figure 7.10 (Inside Receiving state - Final model)

7.3.5 Use Cases and Generated MSCs for the Final Model

We will give only two use case examples for the validation of the final model. The first one will validate the newly added functionality of receiving a fax. We will check the normal sequence of message interactions when the FPS2000 receives a fax from the DestinationFax machine.

The second use case will include both sending and receiving processes. We will describe what happens when the FPS2000 is communicating with the DestinationFax and receives a connectionRequest from the ThirdFax. There are two possible cases. The FPS2000 can either receive the ConnectionRequest before a connection is made with the DestinationFax or after, as described in the requirements. In this use case we will describe two cases when the FPS2000 receives a connectionRequest from the ThirdFax before a connection is made with the DestinationFax, one when it is waiting for digits to be entered and another one when it is waiting for the Go button to be pressed. We will describe only one case when it receives the connectionRequest after a connection is made.

UseCaseReceivingAFax

commDestFax?conReq

^^commDestFax!ring

^^^^commDestFax!answer

^^^^^^commDestFax?data

^^^^^^^^userDisplay!receiving

^^^^^^^^^^userOutPaper!paperOut

^^^^^^^^^^^^commDestFax?endTrans

^^^^^^^^^^^^^^commDestFax!transOK

^^^^^^^^^^^^^^^^commDestFax?disconnect

^^^^^^^^^^^^^^^^^^userDisplay!receptionOK

^^^^^^^^^^^^^^^^^^^^userDisplay!time

%%

Generated MSC for use case "ReceivingAFax"

```
MSC 1;
        INSTANCE ActorUnderTest;
                IN conReq FROM commDestFax;
                OUT ring TO commDestFax;
                OUT answer TO commDestFax;
                IN data FROM commDestFax;
                OUT receiving TO userDisplay;
                OUT paperOut TO userOutPaper;
                IN endTrans FROM commDestFax;
                OUT transOK TO commDestFax;
                IN disconnect FROM commDestFax;
                OUT receptionOK TO userDisplay;
                OUT time TO userDisplay;
        ENDINSTANCE;
ENDMSC;
```

UseCaseRecvConReqFromThirdFax

userPaper?paperIn

^^userDisplay!documentReady

^^^^+RecvConReqBeforeConnect

^^^^userDigit?digitIn

^^^^^^userDisplay!digitDisplay

^^^^^^^^+RecvConReqBeforeConnect

^^^^^^^^userGo?goPressed

^^^^^^^^^^commDestFax!conReq

^^^^^^^^^^^^userDisplay!dialing

^^^^^^^^^^^^^^commDestFax?ring

^^^^^^^^^^^^^^^^commDestFax?answer

^^^^^^^^^^^^^^^^^^+RecvConReqAfterConnect

%%

RecvConReqBeforeConnect

commThirdFax?conReq

^^commThirdFax!ring

^^^^commThirdFax!answer

%%

RecvConReqAfterConnect

commThirdFax?conReq

^^commThirdFax!busy

%%

Generated MSC for use case "RecvConReqFromThirdFax"

MSC 1;

INSTANCE AUT;

IN paperIn FROM userPaper;

OUT documentReady TO userDisplay;

IN conReq FROM commThirdFax;

OUT ring TO commThirdFax;

OUT answer TO commThirdFax;

```
        ENDINSTANCE;
ENDMSC;

MSC 2;
        INSTANCE AUT;
                IN paperIn FROM userPaper;
                OUT documentReady TO userDisplay;
                IN digitIn FROM userDigit;
                OUT digitDisplay TO userDisplay;
                IN conReq FROM commThirdFax;
                OUT ring TO commThirdFax;
                OUT answer TO commThirdFax;
        ENDINSTANCE;
ENDMSC;

MSC 3;
        INSTANCE AUT;
                IN paperIn FROM userPaper;
                OUT documentReady TO userDisplay;
                IN digitIn FROM userDigit;
                OUT digitDisplay TO userDisplay;
                IN goPressed FROM userGo;
                OUT conReq TO commDestFax;
                OUT dialing TO userDisplay;
                IN ring FROM commDestFax;
                IN answer FROM commDestFax;
                IN conReq FROM commThirdFax;
                OUT busy TO commThirdFax;
        ENDINSTANCE;
ENDMSC;
```

7.3.6 Validation of the Final Model

As for the initial model, the validation process for the final model was done manually, using the technique described in section 5.2.5. All the requirements were

expressed in UCTs, as in the examples, and the generated MSCs used to drive the final model. There were some major changes to be made and some tests were iterated a few times. The tests done for the initial model had to be repeated to check the proper functioning of the two processes together. Here too, some modifications had to be made. Finally, when all the tests were done successfully, we could conclude that the model was complete with respect to the requirements expressed in the sending and receiving processes.

We had a finite set of requirements for the fax system and no modifications were made to them. Hence, checking for completeness was not too difficult. In practice however, this is not usually the case since requirements are constantly being changed.

In this chapter a case study was done to assess the effectiveness and feasibility of the use case driven incremental iterative validation strategy. We incrementally developed a fax system using the steps described in chapter 5. The requirements were divided into the two main functions of the fax machine, namely sending and receiving faxes. An initial model of the fax system was built according to the sending process. The final model was an enhancement of the initial model to support the receiving process. Use cases were written for both models and were used to validate them.

This chapter has shown that our validation strategy can be applied in practice and is effective. The validation process is fast and straight forward. The UCT notation helped a lot for structuring the use cases, for example by using SubTrees to reduce their sizes. We have also shown how the different models of the fax system have been checked for completeness. This was possible because we had a finite set of requirements.

Chapter 8

Conclusions and Future Work

In this thesis we introduced a new notation for writing use cases, proposed a use case driven validation framework for incremental development processes and presented a tool for automatically generating message sequence charts (scenarios) from use case trees. To demonstrate our approach, we used the development of a fax system using the ObjecTime toolset as a case study. The results obtained were very promising. The main objective of this thesis which was to develop a means of representing and structuring use cases in a way that is useful for the validation of software systems which are built using an incremental development process has been met, but further study needs to be done to adequately validate and optimize the approach.

For example in our proposed use case driven validation framework, we did not talk about use case management which is a very important issue. Use case management is the whole process of writing and choosing use cases for the validation activity. Right now, use cases are written manually using the UCT notation. This can sometimes be too costly and time consuming when dealing with large systems. Errors introduced when manually constructing use cases may have a negative impact on the system validation process. To reduce such errors, we are studying the use of a graphical notation, for example use case maps, to describe the use cases, and then automatically generate use case trees from them. We are also thinking of implementing a tool to automatically group together related MSCs into a single use case tree. This is helpful for converting and compressing requirements which are expressed in MSCs into use case trees.

The choice of use cases is an issue which we will be considering in the near future, namely : "given a large group of use cases, automatically select the minimum subset which

will fully represent the requirements and hence, optimize the validation activity". The goal is to maximize use case coverage by using the minimum number of use cases.

Concerning our notation, further work could be done to add the static part of TTCN. Now, we consider only a subset of the dynamic part of TTCN, the tree notation part, which describes only behaviour. The static part of TTCN, the tabular notation, can be used to define the message types, the ports, the variables and all the other data types that are included in the behaviour description. This will be helpful if UCT is implemented as a real development tool, so that there is a common definition of data types for both the use cases and the model. We are also thinking of considering the new TTCN96 notation which deals with concurrency to update UCT. This will compare to the existing parallel decomposition capabilities of MSC96.

The idea of relating a use case to a tree is innovative. This has led to the development of a new notation, which can be used to describe the behaviour of systems and also to validate them. The method is very powerful, since it is based on an industrial strength, standard testing notation, TTCN. We hope that our work will contribute to this interesting and ever evolving domain which is software engineering.

Glossary

Actor	The name given to a component of a system which has a structure and a behaviour. The term is also used to define the user of a system.
Alternative Course	Term used by Jacobson to describe the sequence of message interactions in a use case which is different from the normal case.
Ambiguous	A software requirement specification is ambiguous any requirement stated therein has more than one interpretations.
AUT	Actor Under Test.
Basic course	Term used by Jacobson to describe the normal sequence of message interactions in a use case.
BNF	Backus-Naur Form, a metalanguage used to define other languages.
Completeness	the degree to which full implementation of required function has been achieved.
Consistency	the use of uniform design and documentation techniques throughout the software development projects.
Correctness	the extent to which a program satisfies its specification and fulfills the customer's objectives.
Exceptional Scenario	A sequence of message interactions that we would normally obtain in exceptional or error conditions.
FPS2000	Fax Plain and Simple.
Failure	Occurs when a program misbehaves.
HMSC	High level Message Sequence Chart, for compacting MSCs.
High risk scenario	An exceptional scenario which may be very costly if it occurs in the behaviour of a sytem. It has to be tested.
Lex	Lexical analyser, used together with YACC to build compilers.
Low risk scenario	An exceptional scenario which has lesser consequences if it occurs in the behaviour of a system. It may or may not be tested.

Model	A representation of an artifact or activity intended to explain the bahaviour of some aspects of it. The model is less complex or complete than the activity or artifact modeled.
MSC	Message Sequence Chart, a textual and graphical representation of chronological sequences of messages sent between system components and their environment.
Natural language	A language spoken by people, as opposed to a formal language or a language used by computers.
Normal Scenario	A sequence of message interactions that we would normally obtain for a particular scenario when there are no errors.
Object-oriented	A software engineering methodology in which the system is viewed as a collection of objects, attributes of the objects, and operations on the objects, with messages passed from object to object.
OMT	Object Modeling Technique, an object-oriented technique for software development developed by Rumbaugh.
PCO	Point of Control and Observation, any point relative to an actor from which we can control it by inputting certain values and, at the same time, observe the output.
ROOM	Real-time Object Oriented Modeling, a language for specifying, designing, executing and testing real-time systems.
Scenario	A single sequence of interactions between entities in a system.
SDL	Specification Description Language, intended for the specification of complex, event-driven and real-time applications involving many concurrent activities which communicate using discrete signals.
SQA	Software Quality Assurance. In software engineering, a planned and systematic pattern of all actions necessary to provide adequate confidence that the software conforms to established requirements.
Test Case	Input and expected output used for testing.
TTCN	Tree and Tabular Combined Notation, used for the specification of tests for communicating systems.

UCM	Use Case Map, a visual notation for representing use cases.
UCT	Use Case Tree, a new notation for describing use cases in the form of trees.
UML	Unified Modeling Language, a new language for specifying, constructing, visualizing and documenting the artifacts of a software system, and developed by Booch, Rumbaugh and Jacobson.
Use Case	A collection of sequences of interactions between the entities of a system. A use case is one or more scenarios. Usually a use case is one basic course (normal scenario) and several alternative courses (exceptional scenarios).
Validation	Refers to the set of activities that ensure that the software built satisfies the requirements.
Verification	Refers to the set of activities that ensure that the software correctly implements a specific function.
YACC	Yet Another Compiler Compiler, a grammatical analyser used to build compilers.

References

[Amyot97] D. Amyot, "Group Communication Server: A scenario-based design exercise", Technical Report, Department of Computer Science, University of Ottawa, 1997.

[Ander95] M. Andersson, J. Bergstrand, "Formalizing use cases with message sequence charts", Master's Thesis, Dept. of Communication Systems at Lund Institute of Technology, May 1995.

[Baum94] B. Baumgarten, A. Giessler, "OSI Conformance Testing Methodology and TTCN ", Elsevier Science B. V., North-Holland, 1994.

[Beizer90] B. Beizer, "Software Testing Techniques", second edition, Van Nostrand Reinhold, 1990.

[Benhaj97] H. Benhajla, "Traceability in Object-Oriented Quality Engineering - A basis for Regression Analysis of Object-Oriented Software", Master Thesis, University of Ottawa, 1997.

[Boehm76] B.W. Boehm, "Software engineering", IEEE Trans. Comput., vol. c-25, December 1976.

[Boehm84] B.W. Boehm, "Verifying and validating software requirements and design specifications", IEEE Software, volume1, number1, pages 75-88, January 1984.

[Boehm88] B.W. Boehm, "A spiral model of software development and enhancement", System and Software Requirements Engineering, IEEE Computer Tutorial, pages 513-527, May 1988.

[Booch91] G. Booch, "Object-Oriented Design with Applications", The Benjamin Cummings Publ. Co., 1991.

[Buhr96] R.J.A. Buhr, R.S. Casselman, "Use Case Maps for Object-Oriented Systems", Prentice-Hall, 1996.

[Cobb85] R.H. Cobb, "In praise of 4GLs", Datamation, July 1985.

[GEODE95] "ObjectGEODE, The Object Oriented Real-time Application Development Solution", Verilog Inc., Toulouse, France, 1995.

[Goma90] H. Gomaa, "The impact of prototyping on software system engineering", George Mason University, School of Information Technology, 1990.

[Gott90] B.S. Gottfried, "Theory and problems of programming with C", McGraw-Hill, 1990.

[Grab94] J. Grabowski, "The generation of TTCN test cases from MSCs", University of Berne, Institute of Informatics, Technical Report no. IAM-94-004, 1994.

[Grab96] J. Grabowski, "A proposal for real-time extension of TTCN", TIK report no. 20, August 1996.

[Jacob92] I. Jacobson, "Object-Oriented Software Engineering", Addison-Wesley, 1992.

[Kirani94] S. Kirani, W.T. Tsai, "Specification and Verification of Object-Oriented Programs", PhD. Thesis, University of Minnesota USA, December 1994.

[Knight93] K.G. Knightson, "OSI Conformance Testing : IS 9646 explained", McGraw-Hill, 1993.

[Kroon91] J. Kroon, A. Wiles, "A tutorial on TTCN", 11th International IFIP Symposium on Protocol Specification, Testing and Verification, Stockholm, pages 40-92, June 1991.

[Leue94] S. Leue, "What do message sequence charts mean ?", IFIP Transactions, Communication Systems, 1994.

[Lex86] Sun Microsystems Reference Manuals, Programming Utilities for the Sun Workstation, Chapter 7, lex - a lexical analyzer generator, Sun Microsystems, February 1986.

[Log89] L. Logrippo, R. Guillemot, "Derivation of Useful Execution Trees from LOTOS Specifications by using an Interpreter", Formal Description Techniques, K. Turner, Elsevier Science Publishers, 1989.

[Loidl97] S. Loidl, E. Rudolph, U. Rinkel, "MSC96 and beyond- a critical look". Draft version. Accepted for the eight SDL forum, 1997, France.

[MSC92] ITU-T. Recommendation Z.120, Message Sequence Chart (MSC). ITU-T., Geneva, 1992.

[MSC96] ITU-T. Recommendation Z.120, Message Sequence Chart (MSC). ITU-T., Geneva, 1996.

[Myers79] G.J. Myers, "The Art of Software Testing", John Wiley & Sons, 1979.

[Press82] R.S. Pressman, "Software Engineering - A practitioner's approach", McGraw-Hill, 1982.

[Prob92] R.L. Probert, O. Monkewich, "TTCN: The International Notation for Specifying Tests of Communication Systems", Computer Networks, volume 23, no. 5, 1992.

[Prob94] R.L. Probert, "Software Quality Engineering (SQE) Notes", Department of Computer Science, University of Ottawa, Canada, 1994

[Regnell96] B. Regnell, M. Andersson, J. Bergstrand, "A hierarchical use case model with graphical representation", Proceedings of the second International Symposium on Engineering Computer-Based Systems, March 1996.

[Royce70] W.W. Royce, "Managing the development of large software systems: concepts and techniques", in Proc. WESCON, August 1970.

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall, 1991.

[Selic92] B. Selic, G. Gullekson, J. McGee, I. Engelberg, "ROOM: An object-oriented methodology for developing real-time systems", Fifth International Workshop on Computer-Aided Software Engineering, Montreal, Canada, July 1992.

[Selic94] B. Selic, G. Gullekson, P.T. Ward, "Real-time Object-Oriented Modeling", John Wiley & Sons, 1994.

[Selic96] B. Selic, "Automated generation of test drivers from MSC specs", Technical Report 960514, May 1996.

[Tele97] Telelogic Tau, SAAB Combitech Group, Telelogic, Sweden, 1997

[TTCN92] ISO/IEC 9646-3, Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 3 : The Tree and Tabular Combined Notation, 1992.

[UML97] G. Booch, I. Jacobson, J. Rumbaugh, "Unified Modeling Language for real-time systems design", documentation set version 2.0, 1997.

[Wirth76] N. Wirth, "Algorithm + Data Structures = Programs", Prentice-Hall, 1976.

[YACC86] Sun Microsystems Reference Manuals, Programming Utilities for the Sun Workstation, Chapter 8, YACC - Yet Another Compiler Compiler, Sun Microsystems, February 1986.

Appendix A

BNF rules for UCT

The following notation is used :

::=	production symbol
|	alternative
[...]	optional
{...}	zero or more instances
{...}+	one or more instances
(...)	grouping

ActualPar ::=	Value | VarIdentifier
ActualParList ::=	"(" ActualPar { COMMA ActualPar} ")"
AddOp ::=	"+" | "-" | OR
Assignment ::=	VarIdentifier ":=" Expression
AssignmentList ::=	"(" Assignment {COMMA Assignment} ")"
Attach ::=	"+" TreeIdentifier [ActualParList]
BehaviourDescription ::=	{UseCaseTree}+
BehaviourLine ::=	Line [TypeScenario] NEWLINE
BooleanValue ::=	TRUE | FALSE
CancelTimer ::=	CANCEL TimerIdentifier
Construct ::=	Attach | Repeat
DynamicPart ::=	BehaviourDescription
Event ::=	Otherwise | Timeout | Receive | Send
Expression ::=	SimpleExpression [RelOp SimpleExpression]
Factor ::=	[UnaryOp] Primary
FormalParAndType ::=	FormalParIdentifier { COMMA FormalParIdentifier }
	COLON FormalParType

FormalParIdentifier ::=	Identifier
FormalParList ::=	"(" FormalParAndType { SEMICOLON
	FormalParAndType } ")"
FormalParType ::=	TypeIdentifier
HighRiskScenario ::=	H
Identifier ::=	IDENTIFIER
Indentation ::=	{ "^^" }
Line ::=	Indentation StatementLine
LowRiskScenario ::=	L
MessageIdentifier ::=	Identifier
MultiplyOp ::=	"*" | "/" | AND
NormalScenario ::=	N
Number ::=	NUMBER
Otherwise ::=	PCOIdentifier "?" OTHERWISE
PCOIdentifier ::=	Identifier
Primary ::=	Value | VarIdentifier
Qualifier ::=	"[" Expression "]"
ReadTimer ::=	READTIMER TimerIdentifier "(" VarIdentifier ")"
Receive ::=	PCOIdentifier "?" MessageIdentifier
RelOp ::=	"=" | "<" | ">" | "<>" | ">=" | "<="
Repeat ::=	REPEAT TreeIdentifier [ActualParList] UNTIL Qualifier
Send ::=	PCOIdentifier "!" MessageIdentifier
SimpleExpression ::=	Term { AddOp Term }
StartTimer ::=	START TimerIdentifier "(" TimerValue ")"
StatementLine ::=	(Event [Qualifier] [AssignmentList]) |
	(Qualifier [AssignmentList]) |
	AssignmentList | TimerOp | Construct
Term ::=	Factor { MultiplyOp Factor }
Timeout ::=	"?" TIMEOUT TimerIdentifier
TimerIdentifier ::=	Identifier

TimerOp ::=	StartTimer | CancelTimer | ReadTimer
TimerValue ::=	Number
TreeHeader ::=	TreeIdentifier [FormalParList]
TreeIdentifier ::=	Identifier
TypeIdentifier ::=	Identifier
TypeScenario ::=	NormalScenario | HighRiskScenario | LowRiskScenario
UnaryOp ::=	"+" | "-" | NOT
UseCaseTree ::=	TreeHeader NEWLINE {BehaviourLine}+ ENDTREE
Value ::=	Number | BooleanValue
VarIdentifier ::=	Identifier

Appendix B

The Source Code for the UCT to MSC Tool

The source code for the UCT to MSC tool is written in C, and can be used in the UNIX and DOS environments. It is subdivided into different modules to increase readability and understandability. Each module represents a set of related functionalities. After the compilation of all the modules, an executable program "parse" is produced which can be used to parse UCT constructs. The following is a description of the different modules in the tool.

The module which will control the order of compilation of all the other modules is the "makefile". It also contains all the commands which are needed to compile certain modules. The main module is "main.c". It is the highest level module and contains the calls to the functions which will do the parsing, create the internal structure and generate the MSCs. The module "lexical.l" contains the lexical rules of the UCT notation whereas "parser.y" contains the grammatical rules. The module "create.c" contains all the functions for creating the internal structure of a use case, whereas "generate.c" contains all the functions for generating the MSCs from the internal representation. The module "library.c" contains all the miscellaneous functions, such as tree functions, array functions, and so on, that are used in the other modules. The module "analyse.c" is the one coordinating the parsing, generating error messages whenever the syntax of an input use case tree is incorrect. Finally, the module "ui.c" is the one containing the user interface functions. The modules "library.h", "analyse.h" and "ui.h" contain the declarations for "library.c", "analyse.c" and "ui.c" respectively.

The following sections describe the actual C code of each module. Each module has a header which states its name and its purpose. Note that the keywords in C are represented by bold characters.

```
/*************************************************************************
        Program Name   : lexical.l
        Purpose        : This program is used by lex to generate a lexical analyzer
*************************************************************************/

D0              [0-9]
D1              [1-9]

%{
        extern YYSTYPE yylval;
        yywrap( ){ };
%}
%%

"/*"([^\*\/]|\*[^\/]|[^\*]\/)*"*/"   {/* detect and discard comments */}
({D1}{D0}*)|0           { yylval.ival = atoi(yytext);
                          return NUMBER;
                        }
CANCEL                  { return CANCEL; }
L                       { return L; /* low risk scenario */}
H                       { return H; /* high risk scenario */}
N                       { return N; /* normal scenario */}
OTHERWISE               { return OTHERWISE; }
READTIMER               { return READTIMER; }
REPEAT                  { return REPEAT; }
START                   { return START; }
TIMEOUT                 { return TIMEOUT; }
UNTIL                   { return UNTIL; }
AND                     { return AND; }
OR                      { return OR; }
NOT                     { return NOT; }
TRUE                    { return TRUE; }
FALSE                   { return FALSE; }
":="                    { return ASSIGN; }
"+"                     { return PLUS; }
"-"                     { return MINUS; }
","                     { return COMMA; }
":"                     { return COLON; }
";"                     { return SEMICOLON; }
"*"                     { return ASTERISK; }
"/"                     { return SLASH; }
"!"                     { return EXCLAMATIONMARK; }
"?"                     { return QUESTIONMARK; }
"["                     { return LB; }
"]"                     { return RB; }
"{"                     { return LC; }
"}"                     { return RC; }
"("                     { return LP; }
")"                     { return RP; }
"<"                     { return LT; }
">"                     { return GT; }
```

```
"<="                    { return LTE; }
">="                    { return GTE; }
"="                     { return EQUAL;}
"<>"                    { return UNEQUAL; }
[A-Za-z][A-Za-z0-9]*{   yylval.cval = (char*) strdup ((char*)yytext);
                        return IDENTIFIER;
                    }
[ \t]                   {/* detect and discard whitespace */}
"^"                     {  yylval.ival = 1;
                        return INDENT;
                    }
\n                      { return NEWLINE;}
[%]*[\n]*               { return ENDTREE;}

%%


/*************************************************************************
        Program Name    : parser.y
        Purpose         : This program is used by YACC to generate a parser for the
                          UCT grammar
*************************************************************************/

%{
        #define YYSTYPE type_list
        #include "library.c"
        typeNode *Root ;
        typeNode *Node ;
        typeNode *TreeNode;
        typeTree *TreeList;
        typeTree *NewTree;
        int CountNodeBack;
        int CountLine;
        int CountNode = 1;
%}

%token ASSIGN CANCEL IDENTIFIER OTHERWISE READTIMER REPEAT START
%token AND OR NOT MINUS PLUS TRUE FALSE COMMA COLON SEMICOLON
%token INDENT QUESTIONMARK EXCLAMATIONMARK NUMBER TIMEOUT
%token EQUAL LT GT UNEQUAL GTE LTE ENDTREE NEWLINE ASTERISK
%token LP RP LB RB LC RC N L H UNTIL SLASH

%start DynamicPart
%%

ActualPar
        : Value
        | VarIdentifier
        ;
ActualParList
        : LP ActualPar OptActualPar RP
        ;
```

```
AddOp
        : PLUS
        | MINUS
        | OR
        ;
Assignment
        : VarIdentifier ASSIGN Expression
        ;
AssignmentList
        : LP Assignment OptAssignment RP
        ;
Attach
        : PLUS TreeIdentifier OptActualParList
        {
          Node->pco = $<cval>2;
          Node->typeEvent = 7 ;
        }
        ;
BehaviourDescription
        : OptUseCaseTree
        ;
BehaviourLine
        : Line TypeScenario NEWLINE
        ;
BooleanValue
        : TRUE
        | FALSE
        ;
CancelTimer
        : CANCEL TimerIdentifier
        { Node->typeEvent = 5;
          Node->pco = $<cval>2;
        }
        ;
Construct
        : Attach
        | Repeat
        ;
DynamicPart
        : BehaviourDescription
        ;
Event
        : Otherwise
        { Node->typeEvent =  6 ;
        }
        | Timeout
        { Node->typeEvent = 3 ;
        }
        | Receive
        { Node->typeEvent = 1 ;
        }
        | Send
        { Node->typeEvent = 2 ;
        }
```

```
        ;
Expression
        : SimpleExpression OptSimpleExpression
        ;
Factor
        : OptUnaryOp Primary
        ;
FormalParAndType
        : FormalParIdentifier OptFormalParIdentifier COLON FormalParType
        ;
FormalParIdentifier
        : Identifier
        ;
FormalParType
        : TypeIdentifier
        ;
HighRiskScenario
        : H
        ;
Identifier
        : IDENTIFIER
        ;
Indentation
        : OptIndent
        { $<ival>$ = $<ival>1;
          CountLine = CountLine + 1;
          if ((Node == Root) && ($<ival>1 != 0 ))
          {  Node = CreateRight(Node);
             Node->nodeNum = CountNode;
            CountNode = CountNode + 1;
          }
          else if ((Node == Root) && ($<ival>1 == 0) && (CountLine >1))
               {  Node = CreateLeft(Node);
                  Node->nodeNum = CountNode;
                  CountNode = CountNode + 1;
               }
               else if (CountLine > 1)
                    {  if ($<ival>1 == Node->indent)
                       { Node = CreateLeft(Node);
                         Node->nodeNum = CountNode;
                         CountNode = CountNode + 1;
                       }
                       else if ($<ival>1 > Node->indent)
                            { Node = CreateRight(Node);
                              Node->nodeNum = CountNode;
                              CountNode = CountNode + 1;
                            }
                            else
                            { CountNodeBack = (Node->indent) - ($<ival>1);
                              Node = GoBack(Node, CountNodeBack);
                              Node = CreateLeft(Node);
                              Node->nodeNum = CountNode;
                              CountNode = CountNode + 1;
```

```
                        }
                    }
        }
        ;
Line
        : Indentation StatementLine
        { Node->indent = $<ival>1;
        }
        ;
LowRiskScenario
        : L
        ;
MessageIdentifier
        : Identifier
        { Node->message = $<cval>1;
        }
        ;
MultiplyOp
        : ASTERISK
        | SLASH
        | AND
        ;
NormalScenario
        : N
        ;
Number
        : NUMBER
        ;
OptActualPar
        : /*empty*/
        | OptActualPar COMMA ActualPar
        ;
OptActualParList
        : /*empty*/
        | ActualParList
        ;
OptAssignment
        : /*empty*/
        | OptAssignment COMMA Assignment
        ;
OptAssignmentList
        : /*empty*/
        | AssignmentList
        ;
OptBehaviourLine
        : OptBehaviourLine  BehaviourLine
        | BehaviourLine
        ;
OptFactor
        : /*empty*/
        | OptFactor MultiplyOp Factor
        ;
OptFormalParIdentifier
        : /*empty*/
```

```
        | OptFormalParIdentifier COMMA FormalParIdentifier
        ;
OptFormalParList
        : /*empty*/
        | LP FormalParAndType OptFormalParAndType RP
        ;
OptFormalParAndType
        : /*empty*/
        | OptFormalParAndType SEMICOLON FormalParAndType
        ;
OptIndent
        : /*empty*/ { $<ival>$ = 0;}
        | OptIndent INDENT INDENT
        { $<ival>$ = $<ival>1 + $<ival>2 ;
        }
        ;
OptQualifier
        : /*empty*/
        | Qualifier
        ;
OptSimpleExpression
        : /*empty*/
        | RelOp SimpleExpression
        ;
OptTerm
        : /*empty*/
        | OptTerm AddOp Term
        ;
OptUnaryOp
        : /*empty*/
        | UnaryOp
        ;
OptUseCaseTree
        : UseCaseTree
        | OptUseCaseTree UseCaseTree
        ;
Otherwise
        : PCOIdentifier QUESTIONMARK OTHERWISE
        ;
PCOIdentifier
        : Identifier
        { Node->pco = $<cval>1;
        }
        ;
Primary
        : Value
        | VarIdentifier
        ;
Qualifier
        : LB Expression RB
        ;
ReadTimer
        : READTIMER TimerIdentifier LP VarIdentifier RP
        ;
```

```
Receive
        : PCOIdentifier QUESTIONMARK MessageIdentifier
        ;
RelOp
        : EQUAL | LT | GT | UNEQUAL | GTE | LTE
        ;
Repeat
        : REPEAT TreeIdentifier OptActualParList UNTIL Qualifier
        ;
Send
        : PCOIdentifier EXCLAMATIONMARK MessageIdentifier
        ;
SimpleExpression
        : Term OptTerm
        ;
StartTimer
        : START TimerIdentifier LP TimerValue RP
        {  Node->typeEvent = 4;
           Node->pco = $<cval>2;
           Node->value = $<ival>4;
        }
        ;
StatementLine
        : Event OptQualifier OptAssignmentList
        | Qualifier OptAssignmentList
        | AssignmentList
        | TimerOp
        | Construct
        ;
Term
        : Factor OptFactor
        ;
Timeout
        : QUESTIONMARK TIMEOUT TimerIdentifier
        {  Node->pco = $<cval>3;
        }
        ;
TimerIdentifier
        : Identifier
        ;
TimerOp
        : StartTimer
        | CancelTimer
        | ReadTimer
        ;
TimerValue
        : NUMBER
        ;
TreeHeader
        : TreeIdentifier OptFormalParList
        ;
TreeIdentifier
        : Identifier
        ;
```

```
TypeIdentifier
        : Identifier
        ;
TypeScenario
        : /*empty*/
        { Node->typeScenario = 'T';
        }
        | NormalScenario
        { Node->typeScenario = 'N';
        }
        | LowRiskScenario
        { Node->typeScenario = 'L';
        }
        | HighRiskScenario
        { Node->typeScenario = 'H';
        }
        ;
UnaryOp
        : PLUS
        | MINUS
        | NOT
        ;
UseCaseTree
        : TreeHeader
        {  Root = CreateRoot(TreeNode);
           Node = Root;
           CountLine = 0;
           Node->nodeNum = CountNode;
           CountNode = CountNode + 1;
           if (TreeList == NULL)
           { TreeList = CreateTreeList(TreeList);
             TreeList->Name = $<cval>1;
             TreeList->TreeRoot = Root;
           }
           else
           { NewTree = BuildTreeList(TreeList);
             NewTree->Name = $<cval>1;
             NewTree->TreeRoot = Root;
           }
        }
        NEWLINE
        OptBehaviourLine
        ENDTREE
        ;
Value
        : Number
        | BooleanValue
        ;
VarIdentifier
        : Identifier
        ;

%%
```

```
/*************************************************************************
        Program Name    : analyse.h
        Purpose         : Declarations for analyse.c
*************************************************************************/

#ifndef ANALYSE_DEF
#define ANALYSE_DEF
        typedef union
        {  char *cval;
           int ival;
        } type_list;
        #define YYSTYPE type_list
        int Parse();
#endif



/*************************************************************************
        Program Name    : analyse.c
        Purpose         : This is the error-reporting program used by YACC.  Any
                          mistakes in the inputted use case will be reported here
*************************************************************************/

#include <string.h>
#include <stdlib.h>
#include "analyse.h"

extern int yylineno;
int yychar;

#ifdef YYDEBUG
  int yydebug = 1;
#endif

#include "parser.c"
#include "lexical.c"

yyerror (s)
char *s;
{
 fprintf(stderr, "Read from UCT file : %s in line %d : ",s,yylineno);
 switch(yychar)
 {
  case IDENTIFIER:
     fprintf(stderr,"Identifier '%s' not expected.\n",yylval.cval);
     break;
  case 0:
     fprintf(stderr,"Unexpected end of file.\n");
     break;
  default:
     fprintf(stderr,"'%s' not expected.\n",yytext);
     break;
```

```
  }
}

int Parse()
{
  return yyparse();
}


/*************************************************************************
        Program Name   : ui.h
        Purpose        : Declarations for ui.c
*************************************************************************/

#ifndef UI_DEF
#define UI_DEF
#define BUFF_SIZE 256
  int UI_ProcessCmdLine();
  int UI_OpenFiles();
  void UI_CloseFiles();
  int Menu();
  int ChooseNode();
#endif


/*************************************************************************
        Program Name   : ui.c (user interface)
        Purpose        : This program contains all the user interface procedures
*************************************************************************/

#include <stdio.h>
#include "ui.h"
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

static char input_file[BUFF_SIZE] = "";
static char output_file[BUFF_SIZE] = "";
FILE *result;
extern int errno;
extern char* sys_errlist[];

void UI_DisplayUsageInfo()
{
  fprintf(stderr,"-----------------------------------------------------------\n");
  fprintf(stderr,"                UCT_PARSE USAGE INFORMATION\n");
  fprintf(stderr,"-----------------------------------------------------------\n");
  fprintf(stderr," parse [ -i <file_name> ] [ -o <file_name> ]\n\n");
  fprintf(stderr," -i <file_name> specifies input file will be <file_name>\n");
  fprintf(stderr," -o <file_name> specifies output file will be <file_name>\n\n");
  fprintf(stderr," If either or both of -i or -o are missing, then the standard\n");
```

```
  fprintf(stderr,"  input and/or standard output are used. \n\n");
  fprintf(stderr,"  parse -h \n\n");
  fprintf(stderr,"  -h option displays this screen (all other parameters ignored) \n\n");
  fprintf(stderr,"-----------------------------------------------------------\n");
}

int UI_ProcessCmdLine( argc, argv)
int argc;
char *argv[];
{
  int I;  /* current argument number */
  for (i = 1;i < argc; i++)
  {
   if (argv[i][0] == '-')  /* true if we have found a command flag */
   {
    switch (argv[i][1])  /* switch on the flag letter */
    {
     case 'I':
     case 'i':
     {
      /*check if we already have an input file name */
      if (input_file[0] == NULL)
      {
       if (argv[i][2] != NULL)
       {
        /* handles case where there is no space between -i and file name */
        strcpy(input_file,&argv[i][2]);
       }
       /* ++i below advances to next input line argument */
       else if (++i >= argc)
       {
        fprintf(stderr,"UCT_PARSE : Badly specified command line parameters.");
        return -1;
       }
       else
       {
        /* not enough argements in command line */
        strcpy(input_file, argv[i]);
       }
      }
      else
      {
       /* if an input name already exists, there must be a previous -i */
       fprintf(stderr,"UCT_PARSE: Duplicate command line parameter: -%c \n" ,
             argv[i][1]);
       return -1;
      }
      break;
     }
     case 'O':
     case 'o':
     {
      /* check if we already have an output file name */
```

```
      if (output_file[0] == NULL)
      {
       if (argv[i][2] != NULL)
       {
        /* handles case where there is no space between -o and file name */
        strcpy(output_file,&argv[i][2]);
       }
       /* ++i below advances to next input line argument*/
       else if (++i >= argc)
       {
        /* not enough arguments in command line */
        fprintf(stderr,"UCT_PARSE: Badly specified command line parameters.");
        return -1;
       }
       else
       {
        strcpy(output_file,argv[i]);
       }
      }
      else
      {
       /* if an output name already exists, there must be a previous -o */
       fprintf(stderr,"UCT_PARSE: Duplicate command line parameter: -%c \n" ,
            argv[i][1]);
       return -1;
      }
      break;
     }
     case 'H':
     case 'h':
     {
      /* print usage information, and quit */
      UI_DisplayUsageInfo();
      return -1;
     }
     default:
     {
      /* unrecognised command line parameter */
      fprintf(stderr,"UCT_PARSE: Unknown command line parameter: -%c \n",
            argv[i][1]);
      fprintf(stderr,"UCT_PARSE: Type parse -h for usage information \n");
      return -1;
     }
    }
   }
   else
   {
    strcpy(input_file,argv[1]);
   }
  }
  return 0;
}
```

```
int UI_OpenFiles()
{
 result = fopen("result.txt","w");
 if (input_file[0] != NULL)
 {
  if ((yyin = fopen(input_file, "r")) == NULL)
  {
   /* call strerr to get error message for failure reason */
   fprintf(stderr,"UCT_PARSE: Read from input file %s: %s\n",
         input_file, sys_errlist[errno]);
   return -1;
  }
 }
 else
 {
  /* if name is not specified use standard input */
  yyin = stdin;
 }
 if (output_file[0] != NULL)
 {
  if ((yyout = fopen(output_file, "w"))==NULL)
  {
   /* call strerr to get error message for failure reason */
   fprintf(stderr,"UCT_PARSE: Write to output file %s: %s\n",
         output_file, sys_errlist[errno]);
   return -1;
  }
 }
 else
 {
  /* if name is not specified use standard output */
  yyout = stdout;
 }
 return 0;
}

void UI_CloseFiles()
{
 fclose(infp);
 fclose(outfp);
 fclose(result);
}

int Menu()
{
 int choice;

  printf("\n *************** MENU ***************\n");
  printf("1. Generate all MSCs\n");
  printf("2. Generate MSCs containing chosen node\n");
  printf("3. End \n");
  printf("Make a choice please => ");
  scanf("%d",&choice);
```

```
 return(choice);
}

int ChooseNode()
{
 int choice;
 int lineNum = 1;
 FILE *infile;
 char character;
 int  found = 0;

 infile = fopen(input_file,"r");
 printf(" ************ The MainTree ************\n");
 printf("   ");
 character = fgetc(infile);
 while ( !(feof(infile)) && (character != '%') )
 {
  printf("%c",character);
  character = fgetc(infile);
  if ((found == 1) && (character != '%'))
  {  printf("%d. ",lineNum);
     lineNum = lineNum + 1;
     found = 0;
  }
  if (toascii(character) == 10) found = 1;
 }
 printf("\nMake a choice please => ");
 scanf("%d",&choice);
 fclose(infile);
 return(choice);
}



/*************************************************************************
        Program Name   : library.h
        Purpose        : Declarations for library.c
*************************************************************************/

#ifndef LIBRARY_H
#define LIBRARY_H

  struct typeNode
  {
    int nodeNum;
    char state;
    int  indent;
    char *pco;
    char *message;
    char typeScenario;
    int typeEvent;
    int value;
    struct typeNode *Left;
```

```
  struct typeNode *Right;
  struct typeNode *Prev;
 };

 struct typePath
 {
  int Num;
  char typeScenario;
  struct typePath *Next;
  struct typeNode *NodeList;
 };

 struct typeTree
 {
  char *Name;
  int  Count;
  struct typeTree *Next;
  struct typeNode *TreeRoot;
  struct typePath *PathList;
 };

 struct typeColumn
 {
  int value;
  int columnNumber;
  struct typeColumn *Next;
 };

 struct typeRow
 {
  int rowNumber;
  struct typeColumn *column;
  struct typeRow *Next;
 };

 typedef struct typeNode typeNode ;
 typedef struct typeTree typeTree ;
 typedef struct typePath typePath;
 typedef struct typeColumn typeColumn;
 typedef struct typeRow typeRow;
 typePath *CreateFirstPath( );
 typePath *CreateNextPath( );
 typeNode *CreateFirstNode( );
 typeNode *CreateNextNode( );
 typeRow *CreateFirstRow( );
 typeRow *CreateNewRow( );
 typeColumn *CreateFirstColumn( );
 typeColumn *CreateNewColumn( );
#endif
```

```
/*************************************************************************
        Program Name   : library.c
        Purpose        : contains all the functions for creating, reading and deleting
                         nodes, paths and trees
*************************************************************************/

#include "library.h"
#include <malloc.h>
#include <stdio.h>

typeTree *CreateTreeList(Tree)
typeTree *Tree;
{
 if ((Tree = (typeTree *)malloc(sizeof(typeTree))) == NULL)
 {
  printf("NOT ENOUGH MEMORY \n");
  exit(1);
 }
 Tree->Next = NULL;
 Tree->TreeRoot = NULL;
 Tree->PathList = NULL;
 Tree->Count = 0;
 return(Tree);
}

typeTree *BuildTreeList(Tree)
typeTree *Tree;
{
 typeTree *Temp;
 typeTree *NewTree;

 if ((NewTree = (typeTree *)malloc(sizeof(typeTree))) == NULL)
 {
  printf("NOT ENOUGH MEMORY \n");
  exit(1);
 }
 Temp = Tree;
 while (Temp->Next != NULL)
   Temp = Temp->Next;
 Temp->Next = NewTree;
 NewTree->Next = NULL;
 NewTree->TreeRoot = NULL;
 NewTree->PathList = NULL;
 NewTree->Count = 0;
 return(NewTree);
}

typeNode *CreateRoot(Node)
typeNode *Node;
{
 if ((Node = (typeNode *)malloc(sizeof(typeNode))) == NULL)
 {
  printf("NOT ENOUGH MEMORY \n");
```

```
    exit(1);
   }
   Node->Left = NULL;
   Node->Right= NULL;
   Node->Prev = NULL;
   return(Node);
}

typeNode *CreateLeft(Node)
typeNode *Node;
{
   typeNode *Temp;

   if ((Temp = (typeNode *)malloc(sizeof(typeNode))) == NULL)
   {
     printf("NOT ENOUGH MEMORY \n");
     exit(1);
   }
   Temp->Prev = Node;
   Node->Left = Temp;
   Temp->Left = NULL;
   Temp->Right= NULL;
   return(Temp);
}

typeNode *CreateRight(Node)
typeNode *Node;
{
   typeNode *Temp;

   if ((Temp = (typeNode *)malloc(sizeof(typeNode))) == NULL)
   {
     printf("NOT ENOUGH MEMORY \n");
     exit(1);
   }
   Temp->Prev = Node;
   Node->Right= Temp;
   Temp->Left = NULL;
   Temp->Right= NULL;
   return(Temp);
}

typeNode *GoBack(Node, Count)
typeNode *Node;
int      Count;
{
   typeNode *Temp;

   while (Count != 0)
   {
     Temp = Node;
     Node = Node->Prev;
     if (Node->Right == Temp)
```

```
    { Count = Count - 1;}
  }
  return(Node);
}

void DeleteTree(Node)
typeNode *Node;
{
  if (Node != NULL)
  {
    DeleteTree(Node->Left);
    DeleteTree(Node->Right);
    free(Node);
    Node = NULL;
  }
}

void DeleteStructure(TreeList)
typeTree *TreeList;
{ typeTree *TempTree;
  TempTree = TreeList;
  while (TempTree != NULL)
  { DeleteTree(TempTree->TreeRoot);
    TempTree = TempTree->Next;
    free(TreeList);
    TreeList = NULL;
    TreeList = TempTree;
  }
}

void InitializeState(Node)
typeNode *Node;
{
  if (Node != NULL)
  {
    Node->state = 'S';
    InitializeState(Node->Left);
    InitializeState(Node->Right);
  }
}

typePath *CreateFirstPath(TreeList)
typeTree *TreeList;
{
  typePath *TempPath;

  if ((TempPath = (typePath *)malloc(sizeof(typePath))) == NULL)
  {
    printf("NOT ENOUGH MEMORY \n");
    exit(1);
  }
  TreeList->PathList = TempPath;
  TempPath->Next = NULL;
```

```
 TempPath->typeScenario = 'X';
 TempPath->NodeList = NULL;
 return(TempPath);
}

typePath *CreateNextPath(PathList)
typePath *PathList;
{
 typePath *TempPath;
 typeNode *TempNode;

 if ((TempPath = (typePath *)malloc(sizeof(typePath))) == NULL)
 {
    printf("NOT ENOUGH MEMORY \n");
    exit(1);
 }
 PathList->Next = TempPath;
 TempPath->Next = NULL;
 TempPath->typeScenario = 'X';
 TempPath->NodeList = NULL;
 return(TempPath);
}

typeNode *CreateFirstNode(PathList)
typePath *PathList;
{  typeNode *TempNode;

 if ((TempNode = (typeNode *)malloc(sizeof(typeNode))) == NULL)
 {
   printf("NOT ENOUGH MEMORY \n");
   exit(1);
 }
 PathList->NodeList = TempNode;
 TempNode->Right = NULL;
 TempNode->Left = NULL;
 TempNode->Prev = NULL;
 return(TempNode);
}

typeNode *CreateNextNode(Node)
typeNode *Node;
{
 typeNode *TempNode ;
 if ((TempNode = (typeNode *)malloc(sizeof(typeNode))) == NULL)
 {
   printf("NOT ENOUGH MEMORY \n");
   exit(1);
 }
 Node->Right = TempNode;
 TempNode->Right = NULL;
 TempNode->Left = NULL;
 TempNode->Prev = NULL;
 return(TempNode);
```

```
}

typeRow *CreateFirstRow(row)
typeRow *row;
{
 if ((row = (typeRow *)malloc(sizeof(typeRow))) == NULL)
 {
   printf("NOT ENOUGH MEMORY \n");
   exit(1);
 }
 row->rowNumber = 0;
 row->column = NULL;
 row->Next = NULL;
 return(row);
}

typeColumn *CreateFirstColumn(row,val)
typeRow *row;
int val;
{
 typeColumn *tempCol;

 if ((tempCol = (typeColumn *)malloc(sizeof(typeColumn))) == NULL)
 {
   printf("NOT ENOUGH MEMORY \n");
   exit(1);
 }
 row->column = tempCol;
 tempCol->Next = NULL;
 tempCol->value = val;
 tempCol->columnNumber = 0;
 return(tempCol);
}

typeColumn *CreateNewColumn(column,colNum,val)
typeColumn  *column;
int colNum, val;
{
 typeColumn *newColumn;

 if ((newColumn = (typeColumn *)malloc(sizeof(typeColumn))) == NULL)
 {
   printf("NOT ENOUGH MEMORY \n");
   exit(1);
 }
 column->Next = newColumn;
 newColumn->columnNumber = colNum;
 newColumn->value = val;
 newColumn->Next = NULL;
 return(newColumn);
}

typeRow *CreateNewRow(row,rowNum)
```

```
typeRow *row;
int rowNum;
{
 typeRow *newRow;

 if ((newRow = (typeRow *)malloc(sizeof(typeRow))) == NULL)
 {
   printf("NOT ENOUGH MEMORY \n");
   exit(1);
 }
 row->Next = newRow;
 newRow->rowNumber = rowNum;
 newRow->column = NULL;
 newRow->Next = NULL;
 return(newRow);
}



/*************************************************************************
        Program Name   : create.c
        Purpose        : contains the functions for creating the internal structure
                         of a use case
*************************************************************************/

#include <stdio.h>
#include <string.h>


/*  The purpose of the function CreatePathList is to read a binary tree and find
all the possible paths.  These paths, stored in a double linked list, are then
connected to the main linked list, to which each binary tree is already connected.
The number of paths in each binary tree is stored in the main linked list   */
void CreatePathList(typeTree *TreeList)
{
 typeNode *TempNode;
 typePath *PathList;
 typeNode *PathNode;
 typeNode *TreeNode;
 int     End;
 int     CountPath = 1;
 int     CountNode = 1;
 int     Finish = 0;

 TreeNode = TreeList->TreeRoot;
 InitializeState(TreeNode);
 PathList = CreateFirstPath(TreeList);
 PathList->Num = CountPath;
 PathNode = CreateFirstNode(PathList);
 TempNode = TreeNode;
 while (Finish != 1)
 { if (TreeNode->Right != NULL)
    { while (TreeNode->Right->state != 'X')
```

```
{  TempNode = TreeNode;
   if (CountPath != 1)
   {  PathList = CreateNextPath(PathList);
      PathList->Num = CountPath;
      PathNode = CreateFirstNode(PathList);
      CountNode = 1;
   }
   CountPath = CountPath + 1;
   while (TempNode->Right != NULL)
   {  if (TempNode->Right->state != 'X')
      {  while (TempNode->Right != NULL)
         {  if (TempNode->state != 'Y')
            {  if (CountNode != 1)
               {  PathNode = CreateNextNode(PathNode);
               }
               if ( TempNode->typeScenario != 'T')
               {  PathList->typeScenario = TempNode->typeScenario;
               }
               PathNode->numNode = TempNode->numNode;
               PathNode->pco = TempNode->pco;
               PathNode->message = TempNode->message;
               PathNode->typeScenario = TempNode->typeScenario;
               PathNode->typeEvent = TempNode->typeEvent;
               PathNode->value = TempNode->value;
               CountNode = CountNode + 1;
            }
            else
               break;
            TempNode = TempNode->Right;
         }
      }
      while (TempNode->state == 'Y')
      {  TempNode = TempNode->Left;
      }
      if (TempNode->Right == NULL)
      {  if (CountNode != 1)
         {  PathNode = CreateNextNode(PathNode);
         }
         if ( TempNode->typeScenario != 'T')
         {  PathList->typeScenario = TempNode->typeScenario;
         }
         PathNode->numNode = TempNode->numNode;
         PathNode->pco = TempNode->pco;
         PathNode->message = TempNode->message;
         PathNode->typeScenario = TempNode->typeScenario;
         PathNode->typeEvent = TempNode->typeEvent;
         PathNode->value = TempNode->value;
         CountNode = CountNode + 1;
      }
   }
   TempNode->state = 'Y';
   End = 0;
   while ((End != 1) && (TempNode != TreeNode))
```

```
        { if (TempNode->Left == NULL)
          { TempNode->state = 'X';
            TempNode = TempNode->Prev;
          }
          else if (TempNode->Left->state == 'X')
                { TempNode->state = 'X';
                  TempNode = TempNode->Prev;
                }
                else (End = 1);
        }
        TempNode->state = 'Y';
      }
    }
    else
    {
     if (CountPath != 1)
     { PathList = CreateNextPath(PathList);
       PathList->Num = CountPath;
       PathNode = CreateFirstNode(PathList);
       CountNode = 1;
     }
     CountPath = CountPath + 1;
     if ( TempNode->typeScenario != 'T')
     { PathList->typeScenario = TempNode->typeScenario;
     }
     PathNode->numNode = TempNode->numNode;
     PathNode->pco = TempNode->pco;
     PathNode->message = TempNode->message;
     PathNode->typeScenario = TempNode->typeScenario;
     PathNode->typeEvent = TempNode->typeEvent;
     PathNode->value = TempNode->value;
     CountNode = CountNode + 1;
    }
    if (TreeNode->Left != NULL)
    { TreeNode = TreeNode->Left ; TempNode = TreeNode;
    }
    else Finish = 1;
  }
  TreeList->Count = CountPath - 1;
}

/*This function will create a list of path for each binary tree in the main linked list */
void CreateInternalStructure(TreeList)
typeTree *TreeList;
{
  typeTree *TempTree;
  TempTree = TreeList;

  while (TempTree != NULL)
  { CreatePathList(TempTree);
    TempTree = TempTree->Next;
  }
}
```

```
/*************************************************************************
        Program Name   : generate.c
        Purpose        : contains the functions for generating MSCs from the
                         internal structure representation of a use case
*************************************************************************/

#include <stdio.h>
#include <string.h>

int CountMSC = 1;

/* The function FindNumberPath is used to go through the main linked list to find a
matching SubTree. Then the number of paths in the SubTree, which was stored
before, is returned. */
int FindNumberPath(TreeList,Name)
typeTree *TreeList;
char *Name;
{
 typeTree *TempTree;

 if (TreeList->Next == NULL)
 {
  printf("UseCaseTree  %s  is not defined \n",Name);
  exit(1);
 }
 else
 {  TempTree = TreeList->Next;
    while (TempTree != NULL)
    {  if (!(strcmp(Name,TempTree->Name)))
       {  return(TempTree->Count);
       }
       else
         TempTree = TempTree->Next;
    }
 }
 printf("There is a name mismatch for the UseCaseTree %s \n",Name);
 exit(1);
}

/* The function AddSubTreePath is used to add a path from a SubTree to the
NewPath */
typeNode *AddSubTreePath(TreeList,Name,Position,Node,TestNode,NewPathList)
typeTree *TreeList;
char *Name;
int Position;
typeNode *Node;
typeNode *TestNode;
typePath *NewPathList;
{
 typeTree *TempTree;
 typePath *TempPath;
 typeNode *TempNode;
```

```
int Found = 0;

if (TreeList->Next != NULL) TempTree = TreeList->Next;
while ((TempTree != NULL) && !(Found))
{  if (!(strcmp(Name,TempTree->Name)))
   {  Found = 1;
      TempPath = TempTree->PathList;
      while ((TempPath != NULL) && (TempPath->Num != Position))
      TempPath = TempPath->Next;
      if (TempPath->typeScenario != 'X')
      {  NewPathList->typeScenario = TempPath->typeScenario;
      }
      TempNode = TempPath->NodeList;
      while (TempNode != NULL)
      {
        Node->nodeNum = TempNode->nodeNum;
        Node->pco = TempNode->pco;
        Node->message = TempNode->message;
        Node->typeScenario = TempNode->typeScenario;
        Node->typeEvent = TempNode->typeEvent;
        Node->value = TempNode->value;
        if ((TestNode->Right != NULL) || (TempNode->Right != NULL))
          Node = CreateNextNode(Node);
        TempNode = TempNode->Right;
      }
      return(Node);
   }
   else
   TempTree = TempTree->Next;
 }
}

void InitOneDimArray(A, Num)
int A[];
int Num;
{ int Count;
 for(Count=0;Count < Num;Count++)
    A[Count] = 0;
}

int FindValue(combination,r,c)
typeRow *combination;
int r, c;
{
 typeRow *tempRow;
 typeColumn *tempCol;

 tempRow = combination;
 while (tempRow->rowNumber != r)
 {  tempRow = tempRow->Next;
 }
 tempCol = tempRow->column;
 while (tempCol->columnNumber != c)
```

```
  { tempCol = tempCol->Next;
  }
  return(tempCol->value);
}

/* Given a OneDimArray, the function BuildCombination will find all the possible
combinations of the values of the array */
typeRow *BuildCombination(combination,OneDimArray,TotalCom,NumSub)
typeRow *combination;
int OneDimArray[];
int TotalCom;
int NumSub;
{
  int C, Count, Y, X, V, Temp, Prev=1;
  typeRow *tempRow;
  typeColumn *tempCol;

  tempRow = combination;
  for(C=0;C<=NumSub;C++)
  { Count = 0; V = 0;
    while (Count < TotalCom)
    {
      { for(X=1;X<=OneDimArray[C];X++)
        { for(Y=Count;Y<=Count+Prev-1;Y++)
          { if ( Y == 0)
            { tempCol = CreateFirstColumn(tempRow,X);
            }
            else
            {
              tempCol = CreateNewColumn(tempCol,Y,X);
            }
            V = V + 1;
          }
          Count = Count + Prev;
        }
        Temp = V;
    }
    V=0;
  }
  if ( C < NumSub )
  { tempRow = CreateNewRow(tempRow,(C+1));}
    Prev = Temp;
  }
  return(combination);
}

/* The function CheckPath is used to check whether there is a SubTree on a given
path or not  */
int CheckPath(typePath *Path)
{
  typeNode *TempNode;

  TempNode = Path->NodeList;
```

```
 while (TempNode != NULL)
 { if (TempNode->typeEvent == 7)
    { return(1);
    }
    else
      TempNode = TempNode->Right;
 }
 return(0);
}

/* The function PrintMSC is used to convert the information stored in a node into the
corresponding MSC construct */
void PrintMSC(PathNode)
typeNode *PathNode;
{
 switch(PathNode->typeEvent)
 {
   case 1:
     printf("        IN %s FROM %s;\n",PathNode->message,PathNode->pco);
     fprintf(result,"        IN %s FROM %s;\n",PathNode->message,PathNode->pco);
     break;
   case 2:
     printf("        OUT %s TO %s;\n",PathNode->message,PathNode->pco);
     fprintf(result,"        OUT %s TO %s;\n",PathNode->message,PathNode->pco);
     break;
   case 3:
     printf("        timeout %s;\n",PathNode->pco);
     fprintf(result,"        timeout %s;\n",PathNode->pco);
     break;
   case 4:
     printf("        set %s (%i);\n",PathNode->pco,PathNode->value);
     fprintf(result,"        set %s (%i);\n",PathNode->pco,PathNode->value);
     break;
   case 5:
     printf("        reset %s;\n",PathNode->pco);
     fprintf(result,"        reset %s;\n",PathNode->pco);
     break;
   case 6:
     printf("        IN OTHERWISE FROM %s;\n",PathNode->pco);
     fprintf(result,"        IN OTHERWISE FROM %s;\n",PathNode->pco);
     break;
   default:
     printf("        Error;\n");
     fprintf(result,"        Error;\n");
 }
}

/* The function PrintPath is used to print the MSC structure for a given path */
void PrintPath(Path)
typePath *Path;
{
 typeNode *Node;
 typeNode *PrevNode;
```

```
Node = Path->NodeList;
PrevNode = Path->NodeList;
printf("MSC %d",CountMSC);
fprintf(result,"MSC %d",CountMSC);
if (Path->typeScenario == 'N')
{  printf("  ( Normal Scenario );\n");
   fprintf(result,"  ( Normal Scenario );\n");
}
else if (Path->typeScenario == 'H')
     {  printf("  ( High Risk Scenario );\n");
        fprintf(result,"  ( High Risk Scenario );\n");
     }
     else if (Path->typeScenario == 'L')
          {  printf("  ( Low Risk Scenario );\n");
             fprintf(result,"  ( Low Risk Scenario );\n");
          }
          else
           { printf(";\n");
             fprintf(result,";\n");
           }
           printf("      INSTANCE unknown;\n");
           fprintf(result,"      INSTANCE unknown;\n");
           CountMSC = CountMSC + 1;
           free(Path);
           while (Node != NULL)
           {
             PrintMSC(Node);
             PrevNode = PrevNode->Right;
             free(Node);
             Node = PrevNode;
           }
           printf("      ENDINSTANCE;\n");
           printf("ENDMSC;\n\n");
           fprintf(result,"      ENDINSTANCE;\n");
           fprintf(result,"ENDMSC;\n\n");
}

void DeleteCombination(Row)
typeRow *Row;
{
 typeColumn *PrevCol, *Col;
 typeRow *TempRow;

 TempRow = Row;
 while (TempRow != NULL)
 { Col = TempRow->column;
   while (Col != NULL)
   {  PrevCol = Col;
      Col = Col->Next;
      free(PrevCol);
   }
   TempRow = TempRow->Next;
```

```
 }
 TempRow = Row;
 while (Row != NULL)
 {
  Row = Row->Next;
  free(TempRow);
  TempRow = Row;
 }
}

/* This function is used to find all the possible combination paths for a path which
contains one or more calls to SubTrees . It is a recursive function which stops when
there is no SubTree calls on a given path. This path is then printed and deleted. */
void FindAllPath(typePath *Path)
{
 typePath *TempPathList;
 typePath *NewPath;
 typeNode  *TempNode;
 typeNode *NewNode;
 typeNode *Node1;
 typeNode *Node2;
 int  CountSubTree;
 int  CountCombination;
 int  TotalCombination;
 int  ArraySubTree[100];
 typeRow *Combination = NULL;
 int C, CountRow;

 TempPathList = Path;
 TempNode = TempPathList->NodeList;
 CountSubTree = 0;
 CountCombination = 0;
 TotalCombination = 1;
 InitOneDimArray(ArraySubTree,100);
 while (TempNode != NULL)
 { if (TempNode->typeEvent == 7)
    { ArraySubTree[CountSubTree] = FindNumberPath(TreeList,TempNode->pco);
      TotalCombination = TotalCombination * ArraySubTree[CountSubTree];
      CountSubTree = CountSubTree + 1;
    }
    TempNode = TempNode->Right;
 }
 CountSubTree = CountSubTree - 1;
 if (ArraySubTree[0] != 0)
 {
   Combination = CreateFirstRow(Combination);
   Combination = BuildCombination(Combination,ArraySubTree,TotalCombination,
                                  CountSubTree);
 }
 for (C=0;C<TotalCombination;C++)
 { TempNode = TempPathList->NodeList;
    if ((NewPath = (typePath *)malloc(sizeof(typePath))) == NULL)
    {
```

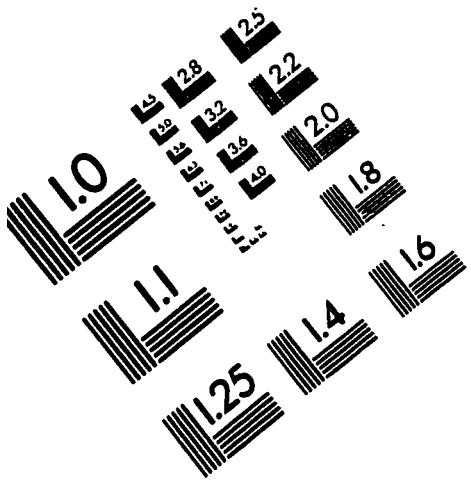
```
        printf("NOT ENOUGH MEMORY \n");
        exit(1);
    }
    NewPath->Next = NULL;
    NewPath->typeScenario = 'X';
    NewPath->NodeList = NULL;
    NewNode = CreateFirstNode(NewPath);
    CountRow = 0;
    NewPath->typeScenario = TempPathList->typeScenario;
    while (TempNode != NULL)
    { if (TempNode->typeEvent == 7)
        { NewNode = AddSubTreePath(TreeList,TempNode->pco,
                                   FindValue(Combination,CountRow,C),
                                   NewNode,TempNode,NewPath);
          CountRow = CountRow + 1;
        }
        else
        { NewNode->pco = TempNode->pco;
          NewNode->nodeNum = TempNode->nodeNum;
          NewNode->message = TempNode->message;
          NewNode->typeScenario = TempNode->typeScenario;
          NewNode->typeEvent = TempNode->typeEvent;
          NewNode->value = TempNode->value;
          if (TempNode->Right != NULL)
              NewNode = CreateNextNode(NewNode);
        }
        TempNode = TempNode->Right;
    }
    if (CheckPath(NewPath))
    {
      FindAllPath(NewPath);
    }
    else
    {
       PrintPath(NewPath);
    }
  }
DeleteCombination(Combination);
}

/* This function is used to generate all the MSCs for each path in the MainTree */
void GenerateAllPath(typeTree *Tree)
{
  typePath *tempPath1;
  typePath *tempPath2;

  tempPath1 = Tree->PathList;
  tempPath2 = Tree->PathList;
  while (tempPath1 != NULL)
  { tempPath2 = tempPath2->Next;
    FindAllPath(tempPath1);
    tempPath1 = tempPath2;
  }
```

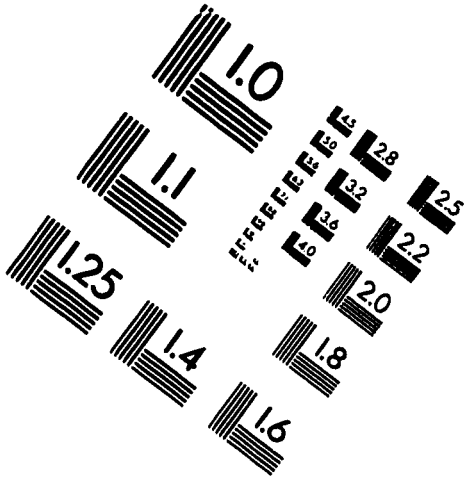
```
 CountMSC = 1;
}

int PathContainNode(Path,Node)
typePath *Path;
int Node;
{
 typeNode *TempNode;

 TempNode = Path->NodeList;
 while (TempNode != NULL)
 { if (TempNode->nodeNum == Node)
    { return(1);
    }
    else
       TempNode = TempNode->Right;
 }
 return(0);
}

/* This function is used to generate the MSCs for each chosen path in the MainTree
*/
void GenerateChosenPath(Tree,nodeChoice)
typeTree *Tree;
int nodeChoice;
{
 typePath *tempPath1;
 typePath *tempPath2;

 tempPath1 = Tree->PathList;
 tempPath2 = Tree->PathList;
 while (tempPath1 != NULL)
 { tempPath2 = tempPath2->Next;
    if (PathContainNode(tempPath1,nodeChoice))
      FindAllPath(tempPath1);
   tempPath1 = tempPath2;
 }
 CountMSC = 1;
}


/*************************************************************************
        Program Name   : main.c
        Purpose        : The main program for the UCT tool. It contains calls to the
                         programs which will parse a use case, create an internal
                         structure and generate MSCs
*************************************************************************/

#include "analyse.c"
#include "ui.c"
#include "create.c"
#include "generate.c"
```

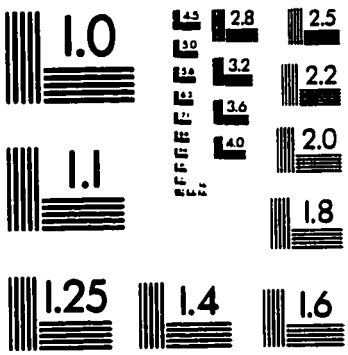
```
#include <stdlib.h>

int main(argc,argv)
int argc;
char *argv[];

{
 int parser_result;
 int choice;
 int End = 0;
 int Node;

 /* process command line arguments */
 if (UI_ProcessCmdLine(argc,argv) != 0)
 {
    return 1;
 }

 /* Open input and output files */
 if (UI_OpenFiles() != 0)
 {
    return 1;
 }

 /* Parse input file */
 parser_result = Parse();

 if (parser_result == 0)
 {
    /*  Creating the internal structure. The procedures are found in create.c  */
    CreateInternalStructure(TreeList);

    do
    {  switch (choice = Menu())
      {
        case 1 : GenerateAllPath(TreeList);
                 break;
        case 2 : Node = ChooseNode();
                 GenerateChosenPath();
                 break;
        default : printf ("\nTHE END");
                  End = 1;
      }
    }
    while (End != 1);

    /*  When all computations are done, the internal structure is deleted  */
    DeleteStructure(TreeList);

 }

 /* Close input and output files */
 UI_CloseFiles();
```

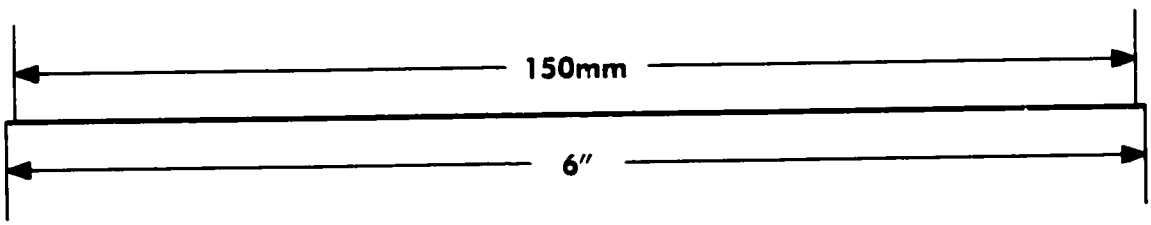
```
 return parser_result;
}


/*************************************************************************
        Program Name    : makefile
        Purpose         : Program that will order the compilation of all the programs
*************************************************************************/

all: parse

parse:  main.c parser.c lexical.c create.c generate.c
        gcc -o parse main.c

parser.c: parser.y
        yacc -d parser.y
        mv y.tab.c parser.c

lexical.c: lexical.l
        lex -t lexical.l > lexical.c
```

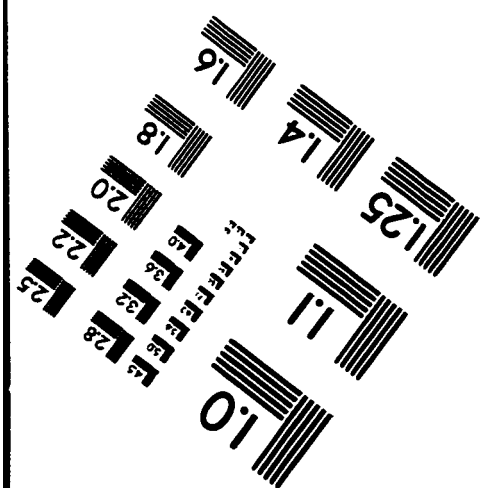
IMAGE EVALUATION
TEST TARGET (QA-3)











APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989