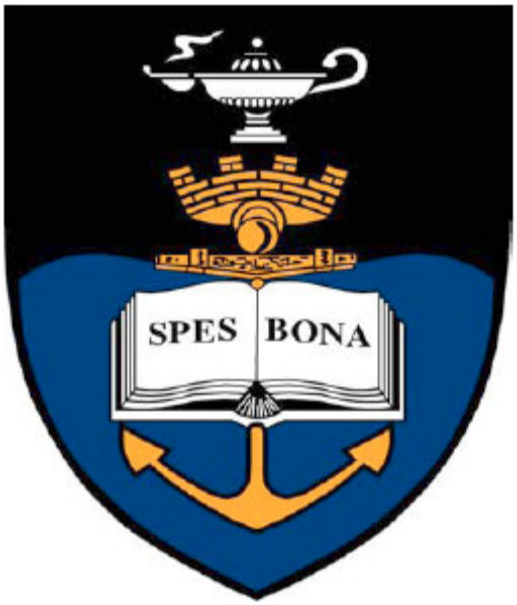
AN ALGORITHMIC STATE MACHINE SIMULATION PACKAGE FOR TEACHING PURPOSES

Prepared by: Joseph Milburn
Supervisor: Samuel Ginsberg



Submitted to the Department of Electrical Engineering in partial fulfilment of the requirements for the degree of Bachelor of Science in Electrical and Computer Engineering at the University of Cape Town

October 2006

Declaration

I Joseph Milburn declare that this thesis, *An ASM Simulation Package for Teaching*, is entirely my own work except for where indicated. All work by other authors has been cited.

This thesis is submitted in partial fulfilment of the requirements for the undergraduate degree of Bachelor of Science in Electrical and Computer Engineering at the University of Cape Town, and has not been submitted to any other university or examining body.

Signed: **Date**: 22 October 2006

_____

Joseph T. Milburn

Acknowledgements

I would like to thank my supervisor Samuel Ginsberg for giving me the opportunity to write this thesis, and for his help and advice, it was much appreciated.

I thank the 3rd year electrical engineering students who helped test the system, particularly Arjun Rhadakrishnan, Gregor George and Qusai Khanbhai.

I would also like to thank my father for all his support and guidance, and of-course for his money without all of which I could never have got this far.

Lastly I would like to thank my close friends at UCT without whose motivation and support I could never have overcome all the memory violations and broken pointers that plagued my existence during this project. In particular I thank Colleen Barry (for her uniquely aggressive motivation style), Dango Mkandawire et al. (for helping me "de-stress"), and Bianca Dolly (for taking me surfing).

Abstract

Introduction

A processing task can be performed by a series of register micro-operations controlled by a sequencing mechanism. The micro-operations can be represented as a hardware algorithm with a series of routines. Deriving the hardware algorithms to perform specific processing tasks is the biggest and most creative challenge in digital logic design. One of the approaches used in overcoming this challenge is the algorithmic state machine (ASM) chart, which has special properties tying it to the hardware implementation of the algorithm it represents.

Objectives

This project aims to develop a simulation environment with a graphical user interface that offers university students grounding in ASM chart theory. The package should:

- Allow ASM elements to be inserted onto an ASM diagram.
- Verify that the constructed ASM diagram is correct.
- Allow elements of the ASM to be associated with inputs and outputs.
- Provide for the inputs to be stimulated and outputs observed while the simulation runs.

ASM Theory

An ASM consists of three types of element:

- States, which represent the stage of execution of the hardware algorithm
- Decision boxes, which direct flow of control to one of multiple exit paths depending on a vector or binary input associated with them
- Conditional output boxes, which follow state boxes and which activate outputs or register operations when the conditions leading to them are met.

Choice of Development Environment

The ASM diagram was implemented using a dynamic array or list of elements, thus C++ was chosen over Java because of its provision for arbitrary memory access. The development environment chosen was wxWindows, because:

- It is very complete.
- Many compilers and platforms are supported: Windows, Linux, Mac, Unix.
- Because of its popularity there is a lot of documentation available for it.
- It is free for personal and commercial use.
- Whenever possible, wxWindows uses the platform SDK (software development kit) resulting in a native look and feel.

Software Design

The Unified Modelling Language, UML, was used to model the software. The software design process entailed specifying use-case models, Class-Responsibility-Collaborator or CRC cards, a class relationship model and object behaviour models (interaction and state diagrams).

Conclusions and Recommendations

A graphical user interface (GUI) allowing an ASM diagram to be built, experimented with and verified was successfully implemented. The simulation package achieves its primary goal of demonstrating ASM's to afford students a clear understanding of how they work.

The feature set of the final simulation package is rather limited. Listed future developments are as follows:

- Allow the diagram canvas to be resized.
- Allow the interface layout to be customized to the user's preference.
- Add state names to the diagram to correspond convention.
- Add a list of elements and variables to the main window.
- Implement register operations and Boolean expressions.
- Provide a file system to allow diagrams to be saved and restored from disk.
- Develop an extension package that develops a digital logic schematic directly from the ASM diagram.

Table of Contents

List of Figures

List of Tables

Glossary

ASM – Algorithmic State Machine.

GUI – Graphical User Interface

PIGUI – Platform Independent Graphical User Interface

UML – Unified Modelling Language

CRC – Class-Responsibility-Collaborator

OO – Object Oriented

SDK – Software Development Kit

CHAPTER 1

INTRODUCTION

1.1 Subject

This thesis project aims to design and implement a software simulation package for algorithmic state machines (ASM's). The simulator will be used as a teaching tool to demonstrate how ASM's work to university students.

1.2 Background

A processing task can be performed by a series of register micro-operations controlled by some sequencing mechanism [1]. The sequence of register micro-operations can in turn be represented as a hardware algorithm with a series of routines or steps that perform the task desired [1]. The data-path and control unit of a digital system can both be specified by such algorithms. The data-path is that part of the digital system which performs arithmetic and data processing operations. The control unit retrieves instructions from the program stored in memory and provides the control signals that manipulate the data-path in order to process those instructions [1]. The data-path and control unit are used in combination with memory to store instructions and data, and various peripherals to perform input and output, and are thus the heart of any digital system.

Deriving the correct hardware algorithms to perform specific processing tasks is the biggest and most creative challenge in digital logic design [1]. One of the approaches used in overcoming this challenge is to specify the procedural steps and decision paths that define a hardware algorithm in a flowchart.

The algorithmic state machine (ASM) chart is a type of flowchart with special properties tying it to the hardware implementation of the algorithm it represents [1]. An ASM describes both the sequence of routines that perform the task and the timing relationship between control unit state changes and the actions taken by the data-path in response to clock pulses [1]. Once an ASM chart has been specified for a particular task, the digital logic required to perform the task can be

developed directly from the chart itself using one flip-flop per state. This decreases design time and effort, which translates to reduced non-recurring engineering costs in the development of a system. To take advantage of this potential saving, it is essential that engineers have a firm grounding in exactly how ASM charts work. This project aims to develop a simulation environment with an intuitive graphical user interface that offers university level students such a grounding through experimentation with ASM's and their functionality.

1.3 Objectives

The simulation package should provide a graphical user interface (GUI) that allows components of ASM's to be inserted onto an ASM diagram workspace. In this way the user should be able to construct and experiment with an ASM. The system should verify that the ASM constructed on the diagram is correct in terms of the standard rules of building an ASM. The user should also be able to associate elements of the ASM with a set of inputs and outputs, such as LED's, buttons, motors and sensors. While the simulator runs the machine the user should be able to stimulate the inputs and observe the resultant outputs to verify that the constructed ASM has been modelled correctly. It is essential that the simulator software be easy to use, that it encapsulates the full functionality of ASM's and that any diagram constructed using the software is verified as being correct if and only if it follows all the rules of ASM's.

1.4 Plan of Development

Chapter 2 examines the theory of ASM's. Each of the elements of ASM's are described, their representation and on an ASM diagram and their functionality explained. ASM Blocks are then explained,

Chapter 3 motivates the programming language and GUI development environment choice made for the implementation. Some of the issues involved in GUI development are also examined.

Chapter 4 reviews the Unified Modelling Language (UML) which was used to model the system. The types of models used in the design are explained. These are use-case specifications, class-responsibility-collaborator cards, class relationship models and object behaviour models.

Chapter 5 presents the project plan and software design is presented. The design consists of the modelling techniques explained in Chapter 4.

Chapter 6 entails the software acceptance testing, which outlines the results of the project, and includes feedback from third year engineering students who tested the software.

Chapter 7 evaluates the project success and suggests future development plans and improvements.

CHAPTER 2

ALGORITHMIC STATE MACHINE THEORY

This chapter of the report aims to describe the theory of ASM's, the elements that they are made up of and how ASMs are timed in response to clock pulses. An example ASM chart is then developed to demonstrate the rules and functionality of ASMs further. Finally the one flip-flop per state method of converting an ASM into physical components that will execute the hardware algorithm is presented.

2.1 Elements of an Algorithmic State Machine

An ASM chart consists of three basic elements: The state box, the scalar decision box and the conditional output box. Vector decision boxes can be added for simplification and convenience[1]. A vector decision box can easily be broken down into multiple scalar decision boxes. Likewise a scalar decision box can be considered a specific type of vector decision box, as shown in the following section.

2.1.1 State Box

A processing task has various states defined which indicate at which stage the control sequence is at any point in time. Such a state is indicated in an ASM diagram by a state box. A state box is represented by a rectangle containing register transfer operations and signals, with a symbolic name in the top left corner [1]. When control enters a particular state, the register transfer operations or output signals can be activated. All signals that are not represented within a particular state are implicitly 0 within that state [1]. In the example state box shown in Fig. 2.1.1 (Mano and Kime, 2004:366) the state is named S1. S1 is assigned a binary code of 000 and contains one register and one signal. Therefore on any clock pulse that occurs while the flow of control is within state S1, the value 7 is transferred into register *R*, and the signal *SIG* is asserted [1]. All other signals are de-asserted.

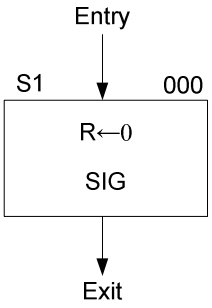


Fig. 2.1.1 State Box

2.1.2 Scalar Decision Box

A scalar decision box is represented by a diamond shape containing a one bit condition [1]. The condition is either a one bit input or a Boolean expression consisting only of inputs. The decision box has two possible exit paths, one path is taken if the condition is true, and the other if it is false [1]. An example decision box is shown in Fig. 2.1.2. (Mano and Kime, 2004:366) When the condition is true, the flow of control of the system follows the right hand exit path, and when the condition is false the left hand path is taken.

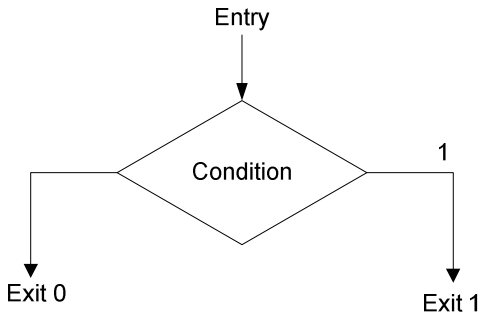


Fig. 2.1.2 Scalar Decision Box

2.1.3 Conditional Output Box

The conditional output box is an oval shaped element with rounded corners, to distinguish it from the state box [1]. The entry path to a conditional output box from a state box must pass through one or more decision boxes [1]. Thus the register operation or output signals inside the conditional output box will be activated only if the conditions specified in the decision boxes leading to the output box are met, hence the term conditional output [1]. Fig. 2.2.3a (Mano and Kime, 2004:366) shows a generic representation of a conditional output box. In Fig. 2.2.3b (Mano and Kime, 2004:366), when the system flow of control reaches the scalar decision box, if the input *IN* is low then control passes to the conditional output box and the output variable *OUT* is set.

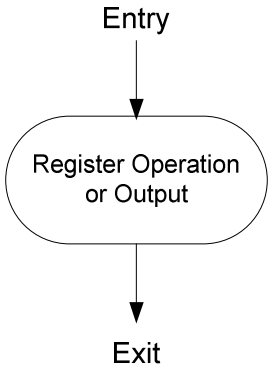


Fig. 2.1.3a Conditional Output Box

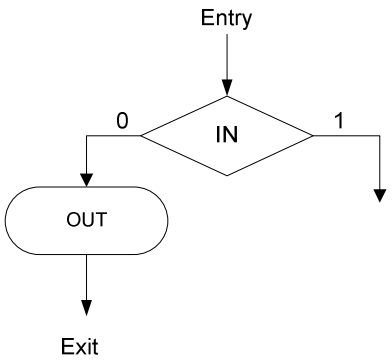


Fig. 2.1.3b Conditional Output Example

2.1.4 Vector Decision Box

The vector decision box serves as an extension to the scalar decision box, allowing an n-bit condition variable to be used for the decision. The condition is made up of n binary inputs or Boolean expressions, each of which must be dependent only on inputs [1]. If n = 1 the element is in effect a scalar decision box, so in the stricter sense n must be greater than 1 for a vector decision box. A vector decision box has 2^n possible exit paths (where n > 1), one for each possible value of the input variable. For the purposes of the ASM simulation package, decision boxes are generalised to vector decision boxes and n may be any integer between 1 and 4 resulting in up to 16 exit paths. A vector decision box is shown in Fig. 2.1.4 (Mano and Kime, 2004:366).

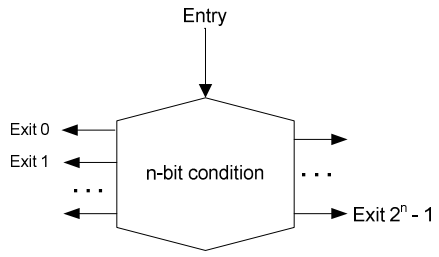


Fig. 2.1.4 Vector Decision Box

2.2 ASM Block

An ASM block consists of a state box, and all the decision and conditional output boxes on the path between the state box and the same or other state boxes [1]. When a clock event occurs within a block:

- All outputs and register operations for which conditions are met are activated.
- Control is transferred to the next state specified by the decision paths leading from the block state. [1]

An example ASM block is shown in Fig. 2.2 (Mano and Kime, 2004:367). In the figure when the flow of control reaches the ASM block, the ASM enters the IDLE

state. This is the block state, while ever the control sequence is within the block the state will be IDLE and the output named *READY* will be asserted. When the next clock pulse occurs, if the input *IN* is 0 then the system stays in the IDLE state. If *IN* is 1 then the register operation R1←0 (clearing register R1) occurs, and one of the next states S0, S1, S2 or S3 is entered depending on the value of the vector Q(1:0) 17.1[1].

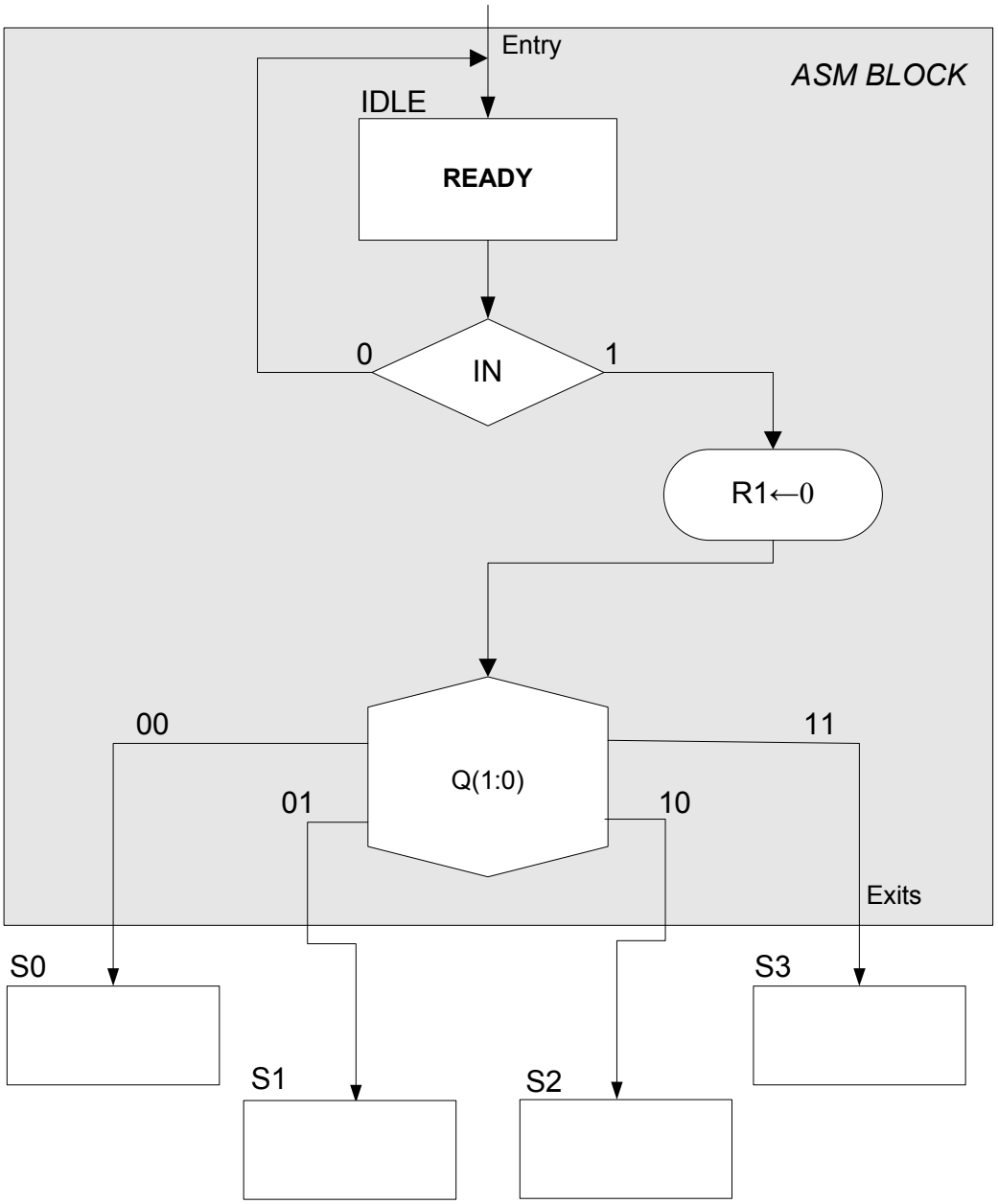


Fig. 2.2 Example ASM Block

2.3 Timing Considerations

To illustrate how the timing of the ASM occurs in response to clock events, the simple example ASM block in Fig. 2.2 will be used. A timing diagram showing what happens to the various signals associated with the block is shown in Fig. 2.3 (Mano and Kime, 2004:368). During clock cycle 1 the ASM is in the IDLE state and so *READY* is asserted. When clock cycle 2 begins, the *IN* input signal is asserted. Assuming that all flip-flops are positive edge triggered, the system remains in the IDLE state since a signal must be high when the clock pulse occurs for it to be recognized as a 1. Therefore when clock cycle 3 begins, the scalar decision box associated with *IN* input takes the exit path leading to the conditional output box, R is cleared, *READY* is de-asserted and the next state is chosen as S1 according to the value of the vector Q(1:0).

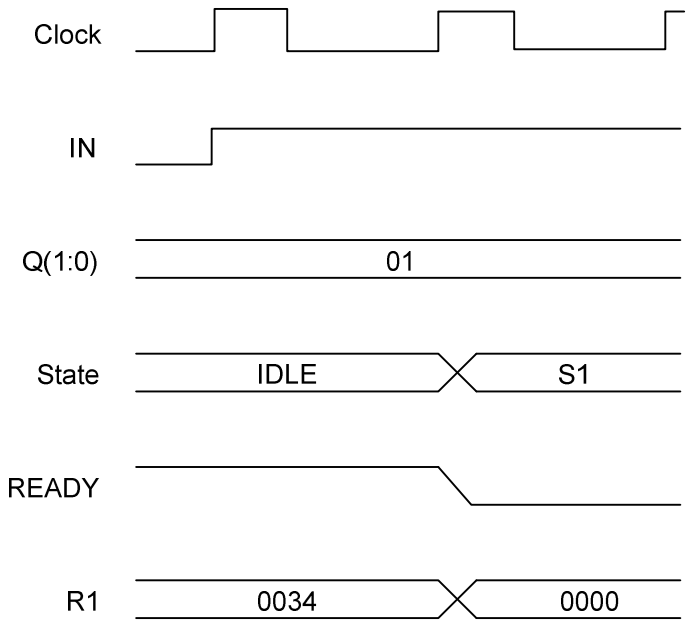


Fig. 2.3 Timing Diagram for ASM Example

2.4 Example ASM

To illustrate the formation of ASM charts the Mano and Kime's binary multiplier example is developed below (2004:369). The multiplier is used to multiply two n-bit unsigned integer values giving a 2n-bit integer result. The hardware algorithm is first developed, then a datapath and control unit for its implementation is proposed, and finally the ASM that describes these two units is derived.

2.4.1 Hardware Algorithm

Multiplying two binary numbers on paper involves successive shifting and adding of the multiplicand, best illustrated by a simple example. As illustrated in Fig. 2.4.1a (Mano and Kime, 2004:369), to carry out the multiplication each successive bit of the multiplier is examined, starting from the least significant bit [1]. If the bit is a one the multiplicand is brought down for the final addition, if the bit is a 0 then all zeros are brought down [1]. Each successive number brought down is left shifted one position from the last.

Decimal	Binary Multiplication	
23	10111	Multiplicand
19	10011	Multiplier
	10111	
	10111	
	00000	
	00000	
	10111	
437	110110101	Product

Fig. 2.4.1a Hand Multiplication Example

When the multiplication procedure is implemented in digital hardware the process changes slightly. In terms of digital circuitry it is less expensive to add the n shifted values that result in the product one at a time instead of all together at the end [1]. Therefore each time the multiplicand or zeros are copied they are immediately added to a partial product which is stored in a temporary register ready for the next addition. Also, instead of shifting the copies of the multiplicand (or all zeros) to the left, the partial products in the register are shifted to the right one position, resulting in the addend and augend being in the same relative position as before [1]. Thus at each successive stage of the multiplication, we

examine the next bit of the multiplier, starting from the least significant bit [1]. If the bit is a 1, we add the multiplicand to the partial product; otherwise there is no need to add all zeros so the partial product is simply shifted to the right. The resultant sum is shifted to the right one position to form the partial product for the next multiplication. The procedure for implementing the "on paper" example from Fig. 2.4.1a is shown in Fig 2.4.1b (Mano and Kime, 2004:370).

Decimal	Binary Multiplication	
23	10111	Multiplicand
19	10011	Multiplier
	00000	Initial Partil Product
	10111	1st bit of multiplier is a 1
	10111	Add
	010111	2nd partial product after right shift
	10111	2nd bit of multiplier is a 1
	(1)000101	Add
	1000101	3rd partial product after shift
	01000101	3rd bit is a 0, so just shift
	001000101	4th bit is a 0
	10111	5th bit is a 1
	110110101	Add
437	0110110101	Shift

Fig. 2.4.1b

Note that the temporary overflow that occurs as a result of the 2nd addition is not a problem since the overflowing (1) is shifted back into the regular most significant position in the next step of the multiplication.

2.4.2 Multiplier Block Diagram

The hardware implementation of the multiplier is shown in Fig. 2.4.2 (Mano and Kime, 2004:371). The algorithm described previously is implemented in the hardware as follows [1]:

- The multiplicand and multiplier are loaded into registers B and Q respectively
- The least significant bit of Q is shifted into the control unit during each stage of the multiplication. This bit is used to determine whether to add A and B then perform a right shift on register A or just perform the shift.

- At the same time the least significant bit of the partial product is shifted into the just most significant position of Q which is now vacant due to the multiplier right shift.
- The binary adder is used to add A and B before each shift.
- The C flip-flop stores any carry be it 0 or 1 after each addition and is reset to nought after each shift
- Counter P keeps count of the number of add and shift operations that have taken place and counts down from n-1 to nought, when the last shift and add occurs.
- The control unit is triggered by the signal G, which activates the multiplication sequence.

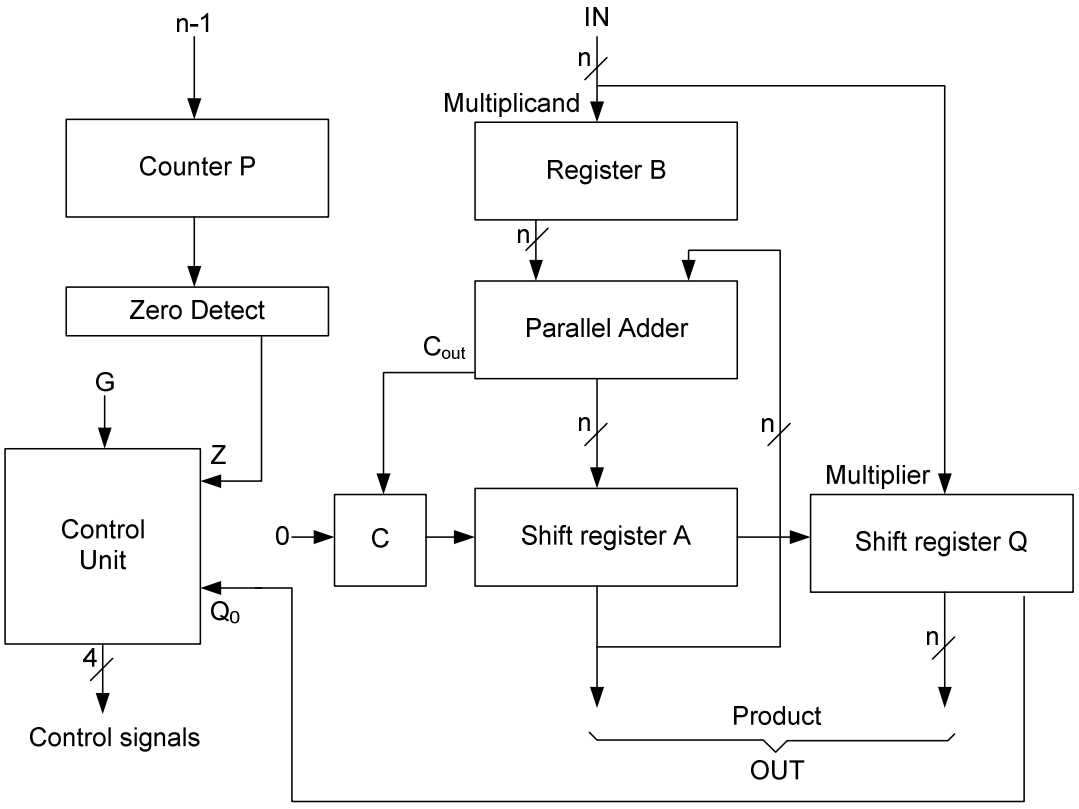


Fig. 2.4.2 Binary Multiplier Block Diagram

2.4.3 Binary Multiplier ASM Chart

An ASM which models the sequence of operations of the binary multiplier is shown in Fig. 2.4.3 (Mano and Kime, 2004:373). The multiplication sequence will only begin if G is 1 on a clock cycle. If this occurs, the one bit carry register C is cleared, as is the partial product register A [1]. We assume that the multiplier and multiplicand are already loaded into registers Q and B respectively. The value n-1 which will result in n shift and add sequences is moved into counter P. Control then moves to MUL0, which is the initial state of the multiplication sequence ASM block. On the next clock pulse, the least significant bit of the multiplier Q_0 is used to decide whether to add A and B and carry any overflow into C_{out} before the state MUL1 [1]. In MUL1 the shift operation occurs, represented by the register transfer expression that follows [1]:

$$C \leftarrow 0,\ C \parallel A \parallel Q \leftarrow \text{sr } C \parallel A \parallel Q$$

The notation C || A || Q simply means a composite register made up of the registers C, A and Q [1]. The notation sr means “shift right”. The details of the register operations are however unimportant in this context. The shift and add sequence will repeat until the counter counts down to zero. This is indicated by the zero detect signal Z being high, which ends the multiplication and returns the system to the idle state until G is asserted again [1].

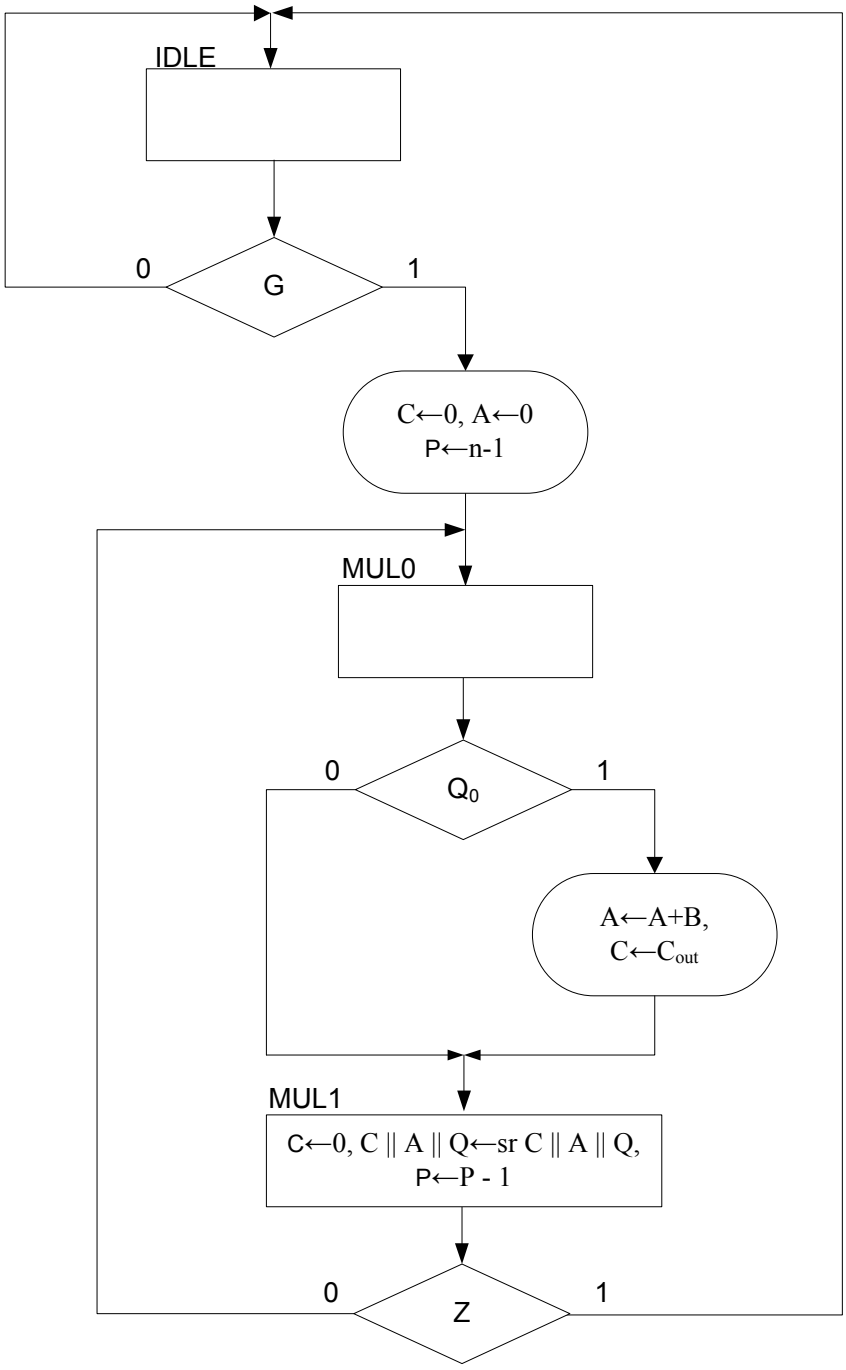


Fig. 2.4.3 ASM Chart for Binary Multiplier

2.5 One Flip-Flop per State

When implementing the control unit of a digital system, two approaches may be used. The first is the use of a sequence register to store control states, and a decoder to provide the output signal corresponding to each of the respective states [1]. This method, while efficient in the number of flip-flops used, involves the development of a state table to design the sequential logic [1]. In order to obtain the input equations for the circuit flip-flops, the state table Boolean functions must be simplified [1]. This requires excessive work, particularly for systems with a large number of states, which is the case for most useful applications.

Another approach involves the use of one flip-flop per state, where the digital logic for the control unit is derived directly from the ASM diagram. This method is far less complex and saves design time for complex systems with a large number of states [1]. The trade-off is the increased number of flip-flops required for the sequential circuit. However the cost of using more flip-flops both in terms of money spent and PCB real estate is likely to be negligible compared to engineer time saved for most systems designs.

The rules for translating ASM diagram components into digital components are shown in Figs. 2.5a to 2.5e (Mano and Kime, 2004:381), and a block diagram of a “one flip-flop per state” control unit for the binary multiplier example is shown in Fig 2.5f (Mano and Kime, 2004:383).

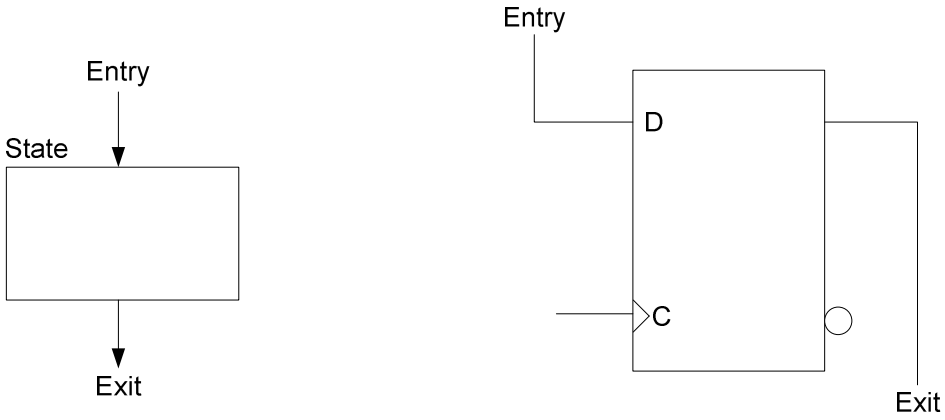


Fig. 2.5a: State box Transformation Rule

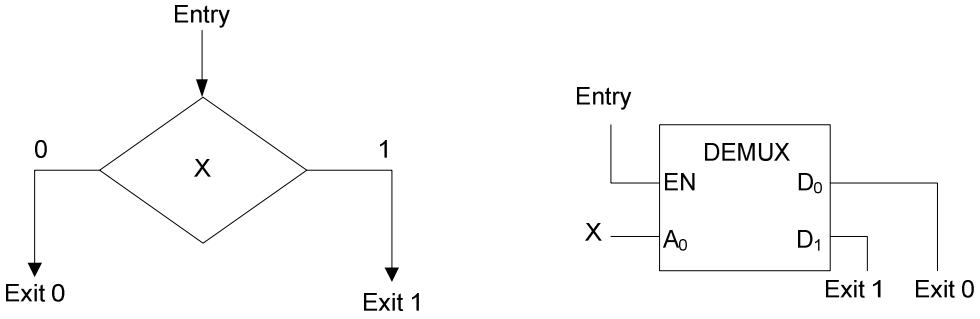


Fig. 2.5b: Scalar Decision Box Transformation Rule

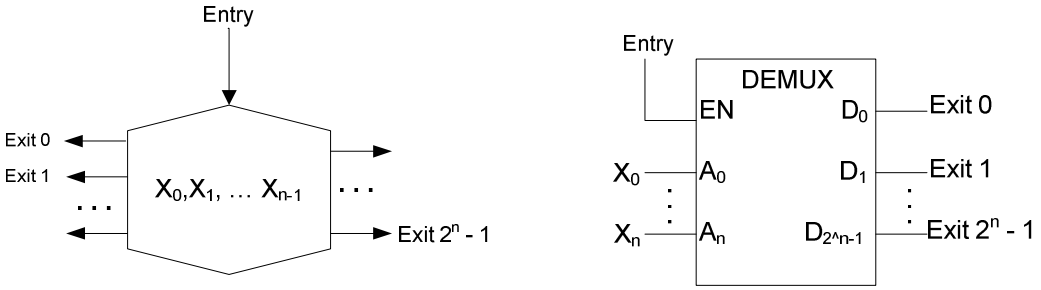


Fig. 2.5c: Vector Decision Box Transformation Rule

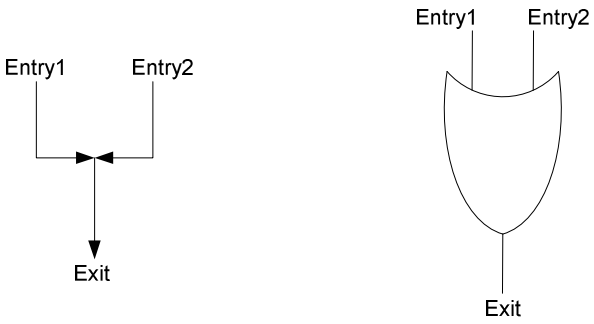


Fig. 2.5d: Junction Transformation Rule

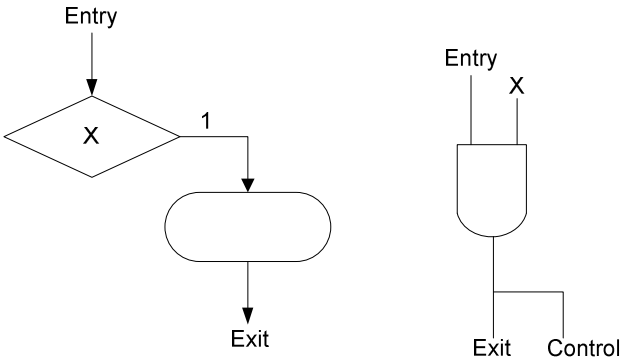


Fig. 2.5e: Conditional Output Transformation Rule

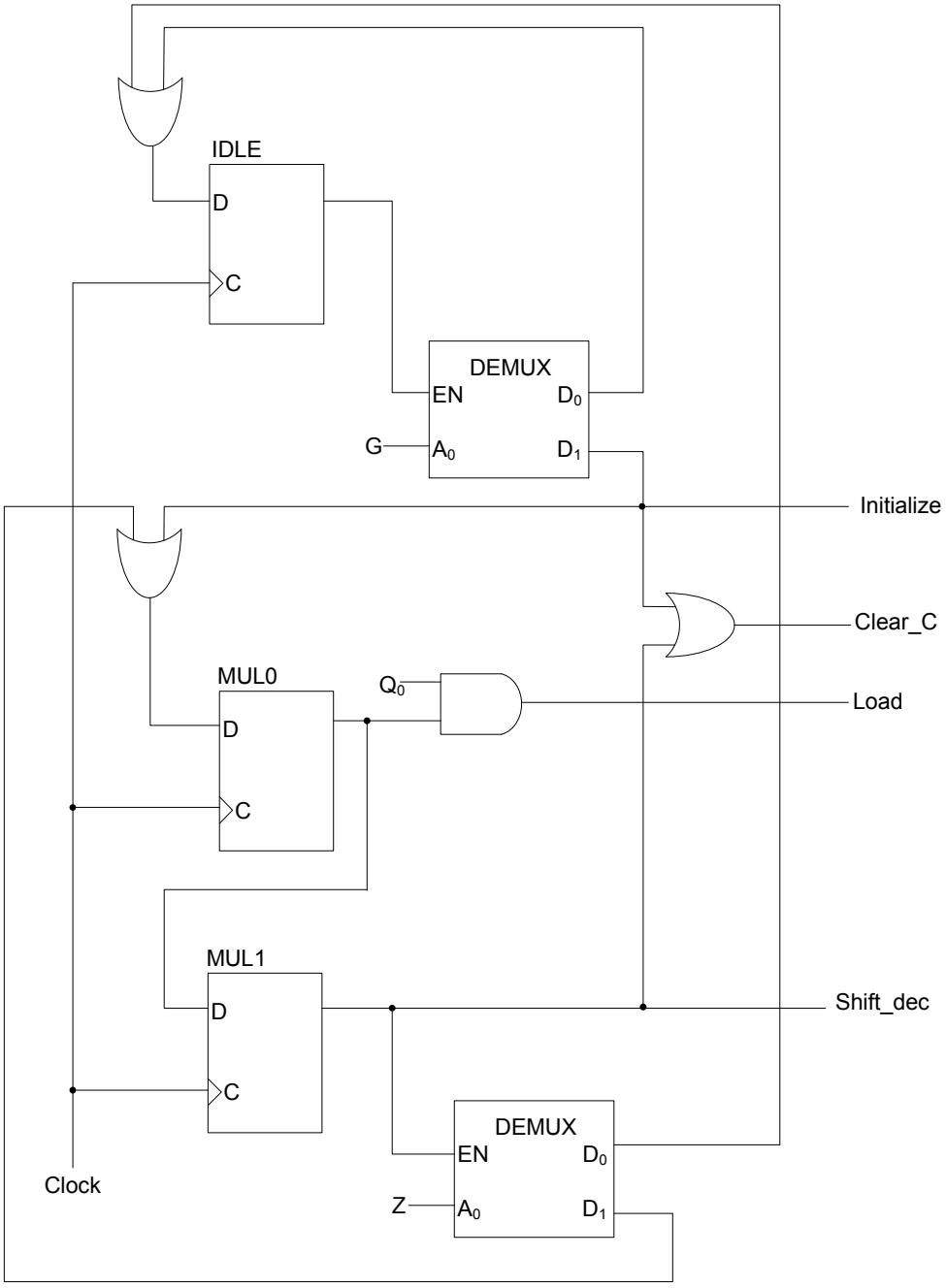


Fig. 2.5f: Binary Multiplier Control Unit with One Flip-flop per State

CHAPTER 3

GUI DEVELOPMENT ENVIRONMENT

This chapter deals with the choice of programming language and development environment for the implementation of the ASM system and GUI. The required features of the development toolkit are outlined, each criterion is then discussed. Finally a GUI development toolkit that closely meets the requirements is chosen

3.1 Introduction

The criteria used to choose the programming language and GUI development environment are as follows:

- Availability: the development environment and compiler should be available for free.
- Ease of development: given the limited timeframe of the project and the inexperience in developing GUIs this was a decisive factor
- Availability of documentation: in the form of online help, tutorials and wizards
- Richness of development kit and programming language features: all the features required of the system and interface should be supported
- Platform Independence: or portability to different target computer systems such as Windows or UNIX.

One motivation for the final criterion was that the UCT computer science department runs exclusively UNIX based machines. If the package were to be used in this department in the future, it would be desirable for the source code to be easily portable to a UNIX platform.

In the following sections some of the issues surrounding GUI development with particular reference to the ASM simulator package are discussed.

3.2 Choice of Programming Language

GUI development toolkits are available for various programming languages including C, C++, Smalltalk, Java, Ada, Tcl, and Python [4]. Most new GUI kits are supported by object-oriented programming languages, because GUI development lends itself well to the OO paradigm [4]. The programming language of choice for most GUI development toolkits is C++, due to the popularity of the language, its flexibility in memory manipulation and pointer use [4].

For the purposes of this thesis project the two programming languages considered were Java and C++, purely because of the programmer's experience in these two languages.

One of the main differences between the two languages was that Java was designed to be easier to use than C++. The Java language focuses on ease of programming where C++ is more execution efficient [3]. Some of the areas in which the two languages differ are:

- Compilation vs Interpretation: C++ is usually compiled where Java uses a mix of interpretation and on the fly compilation. Compilation results in run-time efficiency where interpretation yields quicker compilation [5].
- Efficiency of Generated Code: C++ generates relatively fast, memory efficient executables whereas Java programs have widely reported performance problems [5].
- Automatic memory management: Java has garbage collection, whereas C++ leaves the tedious and error-prone task of reclaiming unused memory to the programmer [5].
- Memory manipulation: C++ allows arbitrary memory access and allows access of objects through the use of pointers where Java allows memory access only through the objects themselves [3].

It is desirable for the simulator package to be execution efficient while providing an intuitive graphical user interface. The easiest and most efficient way to implement the ASM diagram for the package is a dynamically managed list of elements, with each element object pointing to one or more elements in the diagram. Variables can then be associated with each element again through a dynamically allocated list.

This will allow easy manipulation of the diagram components, including adding and deleting elements and variables while maintaining the structure of the ASM chart. This functionality points to the use of C++ rather than Java despite the hassle of manual garbage collection.

3.3 Choice of Development Environment

Apart from the criteria put forth in the introduction to this chapter, the main issue considered when choosing the development toolkit was the method by which controls are placed onto the windows of the program interface.

Placing the controls on their parent window in a layout that is both pleasing to the eye and logical, with similar controls such as buttons or menu items grouped together, is one of the key challenges of GUI development [7].

The layout of the design can be defined in too ways, either by defining the specific positions of controls on a window as parameters to the placing functions in code, or by using a drag and drop development environment [7]. The latter approach allows the programmer to place controls as desired, resize them, specify custom properties etc. This approach is easier and faster but often results in large chunks of code which are often hard to decipher, and therefore defy the “good software engineering” practices of maintainability and reusability [7]. The first approach can be inefficient due to the often tedious and error-prone task of correctly positioning and formatting the controls in code [7].

The GUI development toolkit commonly used in the UCT department of electrical engineering in the past, and one of the most widely used toolkits for C++ is wxWindows. wxWindows uses in code functions for control placement and layout, and has the following advantages that link it to the criteria mentioned above [6]:

- It is very complete. There are many utility classes.
- It is still heavily developed.
- Many compilers and platforms are supported: Windows, Linux, Mac, Unix.
- There is a lot of documentation.
- It is free for personal and commercial use.
- Whenever possible, wxWindows uses the platform SDK (Software Development Kit). This means that a program compiled on Windows will have the look and feel of a Windows program, and when compiled on a Linux machine, it will get the look and feel of a Linux program.

CHAPTER 4

UML THEORY

This chapter reviews the theory and syntax of the Unified Modelling Language. UML is a general modelling system that can be used to model any generic software system. Gain & Kelleher (2004) define UML as "a notational System (including syntax, semantics and pragmatics for its notations) that is principally graphical and aimed at modelling systems using object-oriented concepts."

UML is not a process or methodology, nor is it proprietary. It combines the notations of Booch, Rumbaugh and Jacobson (1997) and was standardized by the OMG (Object Management Group).

It is widely used and has arguably become the standard modelling language for software design. Five types of UML diagrams are described in this chapter. These are:

- Use-case specifications
- Class-responsibility-collaborator cards
- Class relationship models
- Interaction diagrams
- State diagrams

4.1 Use-case specifications

A use case is a sequence of actions performed by a system to achieve some observable result desired by a particular actor. An actor is someone or something outside the system that interacts with it. Fig. 4.1a shows how an actor and use-case is represented in UML. [8]

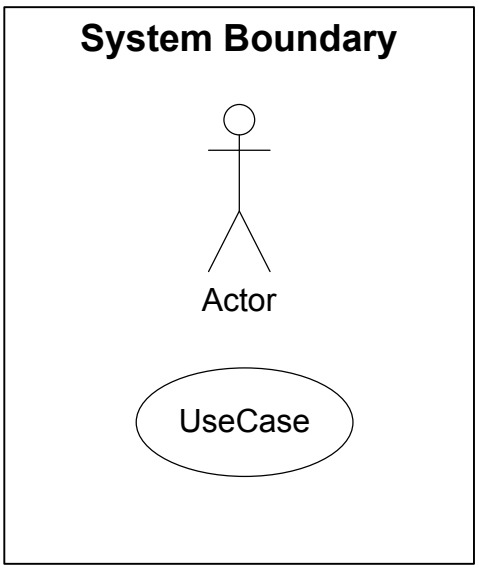


Fig. 4.1a Actor and Use-Case

Use-case specifications present a high level view of the system as seen by outside users of the system without regard for its internal workings. Use-cases model the behaviour of the system as a series of transactions or scenarios which describe the interactions between the user and other external objects with the system itself. Use-cases are used to derive the structural and behavioural models of the system, and to construct test cases which verify the system functionality. Use-case diagrams include [2]:

- Associations: to indicate some interaction between an actor and a use-case. An association is depicted by a solid line, with an optional arrowhead to specify the direction of the invocation of the interaction.
- System boundary boxes: drawn around elements of a use-case diagram to indicate which that those elements are within the scope of a particular system.
- Packages: to organize model elements into groups. Packages are depicted as file folders.

Fig. 4.1b (Ambler, 2004) shows an example use-case diagram where a customer may search for items or place an order using the "Release1" system, or obtain help using the "Release2" system. The "place order" use-case requires the system to interact with the payment processor external system. Similarly the "obtain help" use-case requires interaction with customer service.

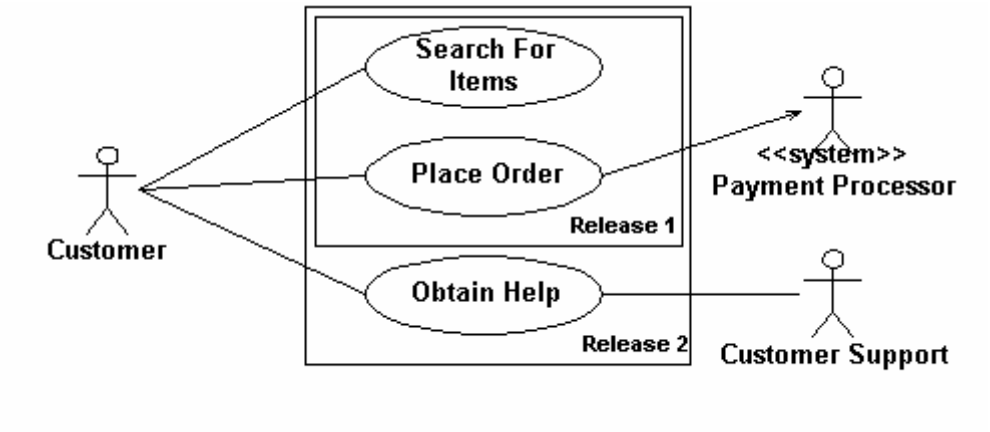


Fig. 4.1b: Example Use-Case Diagram

4.2 Class-Responsibility-Collaborator Cards

CRC cards are a simple way of extracting and laying out the definitions and interactions between classes. They are derived from the system use-case models and define the class hierarchy as well as attributes and methods. [8]

After establishing which of the systems and actors in the use-case models can be accepted as classes, each of the classes is assigned a type. A list of possible class types and examples of objects that might qualify for each type is given below [8].

- External entities – other systems, devices, people
- Objects – reports, displays, signals
- Events – property transfer, completion of a series of robot movements
- Roles – manager, engineer, sales person
- Organizational Units – bank division, group, team
- Places – manufacturing floor, loading dock
- Devices – four-wheeled vehicles or computers
- Interaction (between other objects) – a purchase, a license

The classes are then assigned a set of characteristics which describe the entity they represent. Some of the characteristics that may be assigned are [8]:

- Tangibility: does the class represent a tangible (physical) or abstract (information) entity?
- Inclusiveness: is the class atomic (includes no other classes) or aggregate (has at least one nested object)?
- Sequencing: is the class concurrent (has its own thread of control) or sequential (scheduled by outside resources)?
- Persistence: is the class transient (created and removed during program operation), temporary (created during program operation and removed at termination) or permanent (stored in a database)?

Once the class characteristics have been established each class is assigned a set of responsibilities and collaborators. A class responsibility is a method or attribute of that class. Thus it is the data, objects, behaviour and operations that describe and belong to the class or object it represents. If collaboration exists between two classes, one of the classes needs to send or receive messages from the other in order to fulfil its purpose or carry out its tasks. Examples of types of class collaboration that exist are [8]:

- Aggregation – one class is a member or attribute of the other
- Association – one class must acquire information from the other
- Dependency – some relationship not covered by aggregation or association

4.3 Class Relationship Models

The UML class or object relationship model is a means of modeling a system that emphasizes the structure and attributes of objects, the relationships between them and operations that each performs.

In designing a system one of the key issues to consider is the amount of overall system intelligence that each class or functional entity encapsulates.

Using a large number of simple classes means that each class encapsulates less of the overall system intelligence. Such classes are more reusable, because a small

piece of a system is more likely to be useful to another system than a large chunk of it [2].

Using a small number of complex classes means that each class encapsulates a larger portion of the overall system intelligence. A high degree of encapsulation makes code integration easier. It also means that classes are less likely to be reusable and are more difficult to implement.

As a general rule a class representing an object or entity in the system should have a single well focused purpose. The steps in developing the class relationship model are [8]:

- Create initial design classes
- Define operations, methods and states
- Define attributes
- Define dependencies, associations and generalizations
- Evaluate results

An example of a class diagram is shown in Fig: 4.3a (Gain & Kellher, 2004).

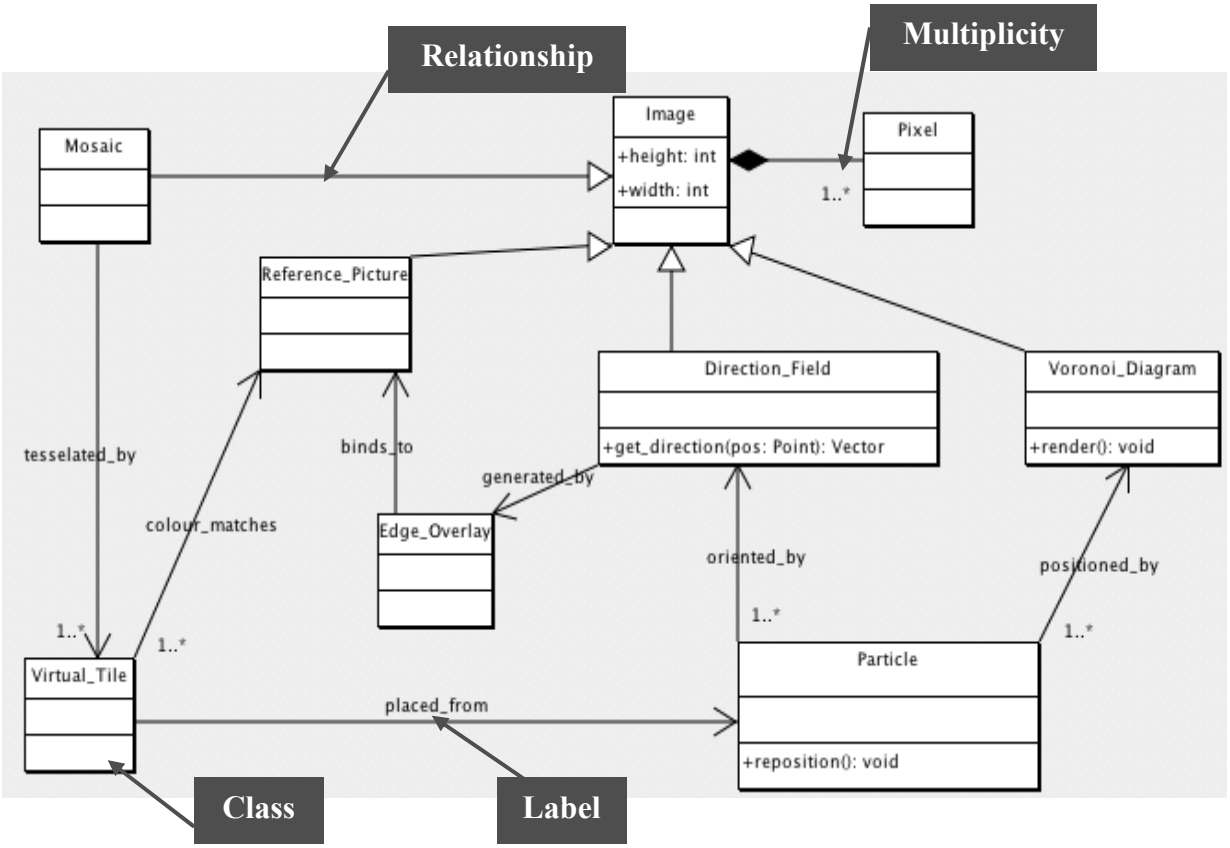


Fig: 4.3a: Example UML Class Diagram

The rectangular objects in the diagram represent classes. Each class has a name field at the top of the rectangle, followed by an attribute field listing the class variables. The third field of a class are the operations of the class or the functions it performs.

A relationship defines some interaction between two classes. Various class relationships exist and are outlined later. Each end of a class relationship may have multiplicity associated with it. This is to indicate the number of objects that may exist on that end of the relationship. The * symbol is used to indicate an infinite number. In the example Fig. 4.3b, either 1 or 2 members of Class A can be associated with 1 or more members of Class B. Table 4.3 (Ambler, 2004) shows the potential multiplicity indicators and their meaning.

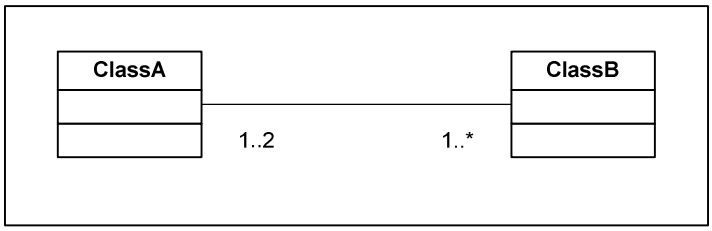


Fig. 4.3b: UML Relationship Multiplicity

Indicator	Meaning
0..1	Zero or one
1	One
0..*	Zero or more
1..*	One or more
n	Exactly n (where n>1)
0..n	Zero to n (n>1)
1..n	One to n (n>1)
n..*	n or more (n>1)
n..m	n to m (n > 1and m > n)

Table 4.3: UML Multiplicity Indicators

A label can be used to indicate further information about the nature of an association. For example in Fig4.3a a particle can be positioned by a Voronoi Diagram.

The various class relationship types available in UML will now be discussed.

4.3.1 Association

An association is the most general type of interaction or relationship between two classes. The roles and multiplicity of the two classes may be indicated on the ends of the line attaching them. Additionally the name of the association may be indicated using a label. The other relationship types are specializations of an association, and so share these properties. An example of a generic association is show in Fig. 4.3.1 (Ambler 2004) where a bank teller serves a customer.

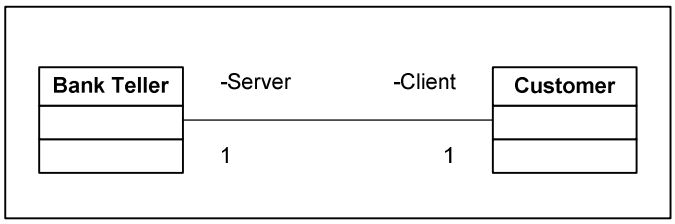


Fig. 4.3.1: UML Class Association

4.3.2 Navigability

Adding an arrowhead to an association implies navigability or direction. In Fig. 4.3.2 (Ambler 2004), a student may enrol for one or more courses. The arrow direction indicates that it is the student that enrols for a course and not vice-versa.

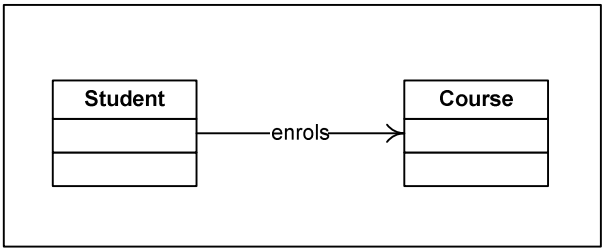


Fig. 4.3.2: UML Class Navigability

4.3.3 Generalisation

A UML generalisation relationship is used to signify inheritance of one class from another. As an example, the car class show in Fig. 4.3.3 is a parent or super class. The subclasses of the car class include a BMW class and a Mercedes class. Subclasses are specific types of the parent class.

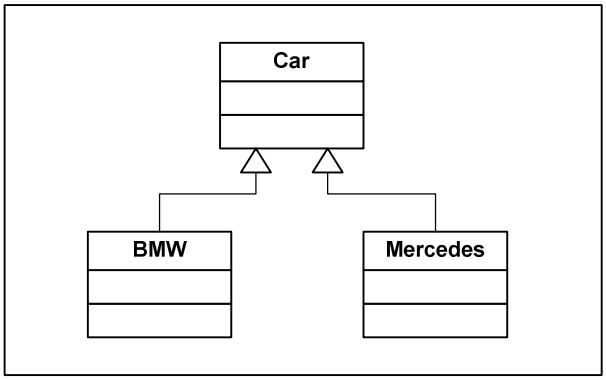


Fig. 4.3.3: UML Class Generalisation

4.3.4 Aggregation

A UML aggregation is used to indicate an ownership [2]. A parent class owns members of a child class. The existence of the parent class is not dependent on the children though, and vice-versa. For example in Fig 4.3.4 a Toyota Taz has passengers and a driver. However if the passengers or driver leave the Taz it continues to exist. If the Taz is destroyed the passengers and driver may become members of another vehicle.

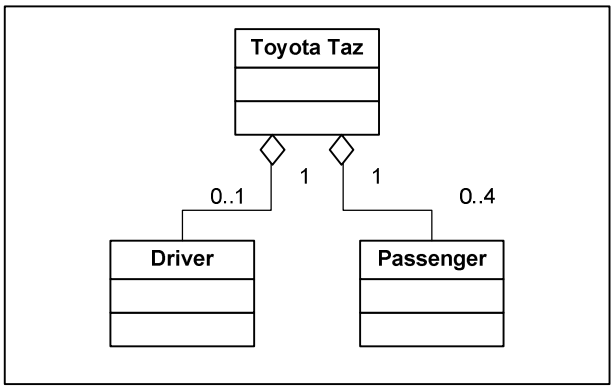


Fig. 4.3.4: UML Class Aggregation

4.3.5 Composition

A UML composition relationship is a stronger form of aggregation where the composite class has responsibility for the creation and destruction of the whole. The composite whole and the parts are coincident, so members of the composite exist for the lifetime of the whole [2]. For example in Fig. 4.3.5a a Toyota Taz has four doors. When a Taz class is created or destroyed it is responsible for creating or disposing of its door class members.

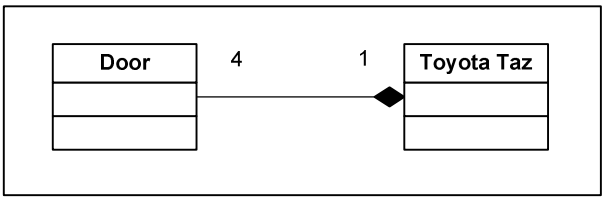


Fig. 4.3.5a: UML Class Composition

Recursive composition may also be used. In Fig. 4.3.5b (Ambler, 2004) a building is made up of one or more rooms, and a room may in turn be composed of one or more smaller rooms.

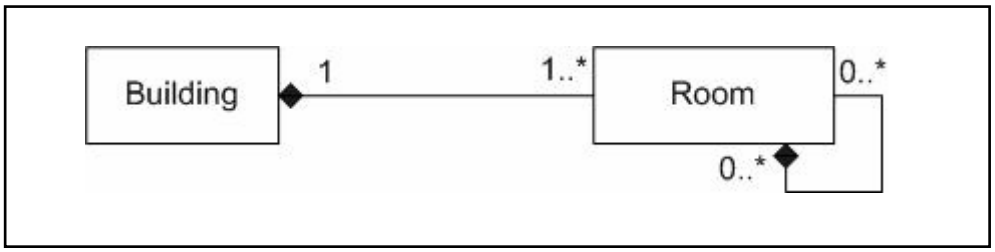


Fig. 4.3.5b: Recursive Composition

4.3.6 Dependency

A dependency relationship suggests that one class (the source) needs the services or functionality of another class (the target). The source has no ownership or control over of the target, which need not even know about the relationship. In Fig. 4.3.6 (Ambler, 2004) the flight simulator class needs input from the joystick class, but the objects of the joystick class can function fully without knowing about the simulator. It is sometimes said that the source class *uses* the target. Dependencies are the least formal of UML associations and are often too broad in meaning to explain the interaction between two classes, and should thus be used sparingly. Vernon suggests that the association relationship is more standard for class interactions not covered by the other UML relationship types (2004).

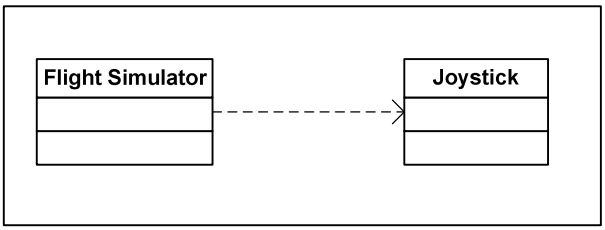


Fig. 4.3.6: UML Class Dependency

4.4 Object Behaviour Models

There are two types of UML object behaviour models. The first is the UML interaction or sequence diagram, and the second is the UML state diagram.

The objective of both of these modelling techniques is to demonstrate the dynamic behaviour of the system, or the sequence of events that take place to achieve the system's various functions. The process involved in developing the object behaviour models is as follows [8]:

- Evaluate use-cases to determine the sequence of system interactions
- Identify the events that sequence the system interaction and relate them to specific objects or components of the system
- Create an interaction diagram for each use-case
- Build a state diagram for important parts of the system
- Object-Oriented Analysis (State and Interaction Diagrams), James Gain

4.4.1 UML Sequence Diagram

A UML sequence diagram is used to model the passing of messages and information that occur during a use-case. As the name suggests a UML sequence diagram shows the sequential logic of a use-case via the ordering of the messages on the diagram [2]. An example sequence diagram is shown in Fig 4.4.1 (Ambler, 2004) which models a student enrolling for a seminar. A message is depicted as a horizontal arrow, pointing in the direction of the flow of information, with a label describing the message being passed. A flow of information in response or reply to a message is indicated by a dashed arrow. The boxes at the top of the diagram depict the actors in the system. The dashed lines hanging from them are the object lifelines, which depict the lifetime of the object in the interaction being modelled. The long thin boxes on top of the lifelines are object activation boxes, and indicate that an object is undertaking some processing task [2].

The sequence of events when a student enrols for a seminar is as follows:

- The student attempts to enrol for a seminar
- The seminar enrolment system queries the course registration system to determine whether the student is eligible to attend the seminar
- The course registration system requests seminar from the student which is sent as a reply
- The course registration system then replies to the eligibility request from the seminar system
- The seminar system replies to the enrolment request from the student

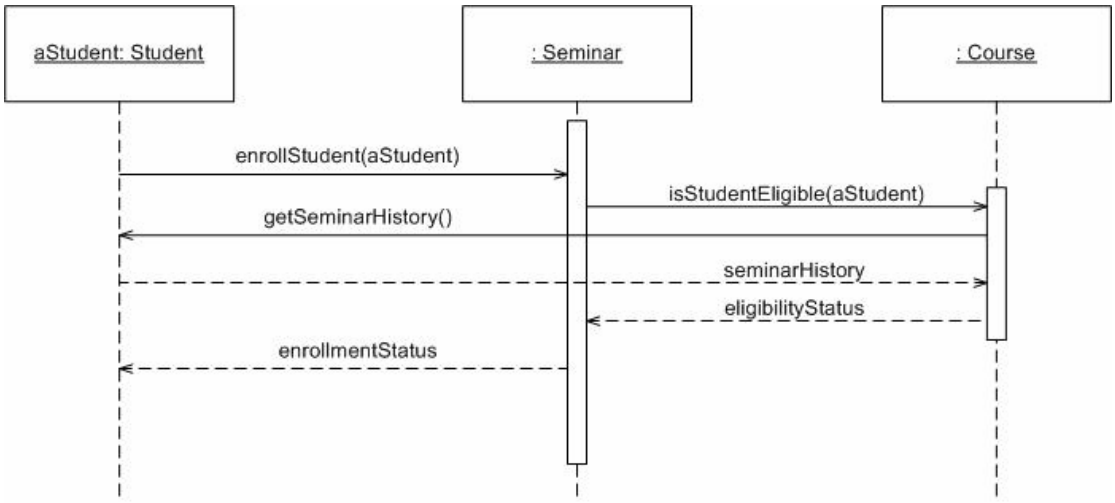


Fig. 4.4.1: Example UML Sequence Diagram

4.4.2 UML State Diagram

State diagrams or state machine diagrams are used to model the behaviour of complex objects or classes. These objects may have various states associated with them representing the stage of execution that the system is in. A state diagram models the various states of the system and the transitions between them. It depicts the events that cause each transition to occur and the actions that are taken when a transition occurs [2].

Fig. 4.4.2 (Ambler 2004) shows an example state diagram. States are represented by the boxes with curved edges. The initial state of the object is indicated by a black circle, and the final state by a bordered circle [2]. Transitions are represented by arrows leading from one state to another, or back to the same state.

A transition label has the format *event [guard]/method list* [2]. The mandatory event label indicates the event that triggers the transition. The optional guard field indicates a condition that must be true for the transition to take place. A list of method invocations may be included if relevant [2]. The methods invoked while within a particular state may also be triggered by events. In Fig.4.4.2, the logSize() method is invoked on entry to the open for enrolment state. In the full state the enrol student event triggers the invocation of the addToWaitingList() method. The syntax for state method invocations is the same as for transitions, except the method list is mandatory and the event is option [2].

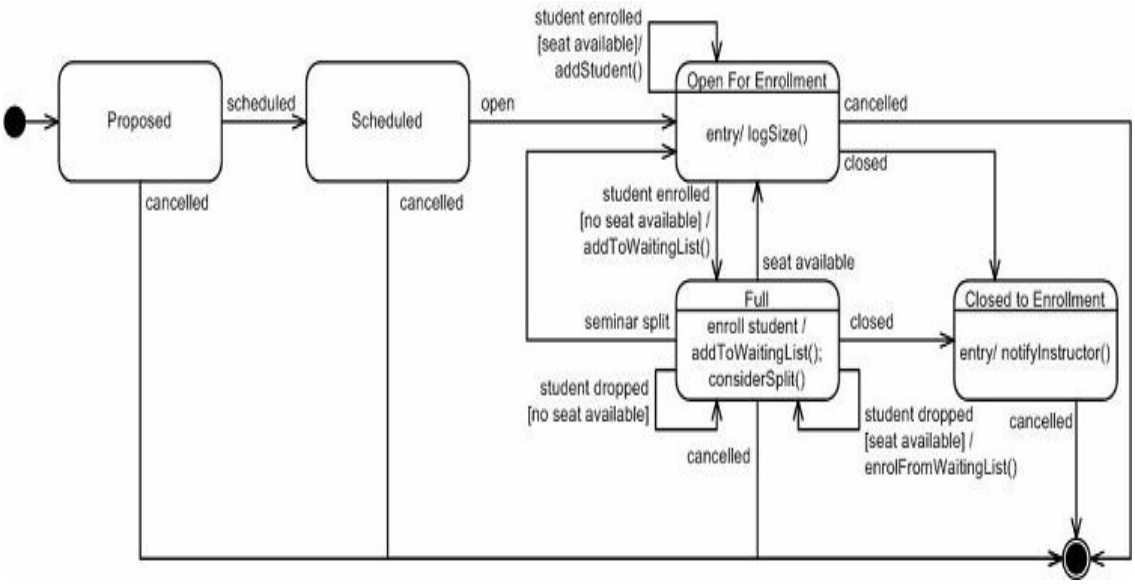


Fig. 4.4.2: Example State Diagram

CHAPTER 5

PROJECT PLAN

This chapter outlines the major functions that the software must achieve and examine the various design issues involved. It goes on to split the software functionality into a set of modules, and outlines what each of these modules should do in relation to the rest of the system.

5.1 Introduction

5.1.1 Major Software Functions

The functional requirements of the ASM simulator software are outlined below. The simulator should:

- Provide an easy to use, intuitive graphical user interface (GUI).
- Allow ASM components to be inserted into a workspace and connected together in order to construct an ASM diagram.
- Verify the correctness of a constructed ASM.
- Run a verified ASM, allowing a user to stimulate inputs and observe the changes in outputs as the simulator runs.
- Step through an ASM element by element and observe what changes occur at each level of machine.
- Save and retrieve constructed ASM's to hard disk or other storage.

5.1.2 Performance and Behaviour Issues

It may take students a number of attempts to construct a correct ASM. Thus it should take a few seconds at most to verify or run an ASM on a University of Cape Town DC or white lab computer.

The simulator must always verify the correctness of an ASM accurately. If an ASM has not been constructed correctly the simulator must recognise the error. This is to ensure that the simulator achieve its primary purpose of teaching students how ASM's work.

5.1.3 Logistic and Technical Constraints

The major constraints or issues of concern of this project are:

- The programmer's lack of experience designing graphical user interfaces.
- The strict time constraints due to the deadline of 23 October 2006.

5.2 Project Estimates

5.2.1 Effort / Difficulty

The design phase of the thesis should be relatively easy. Writing the ASM class code should not present much difficulty either. However implementing the graphical user interface will be a more difficult challenge and will require far greater effort considering the lack of experience in this area.

5.2.2 Time Estimate

The project has been divided into five phases. The initial phase comprises the project plan. The second phase consists of the literature review and software design, and the third phase the prototyping of the ASM classes. The fourth phase will entail developing the GUI and integrating it with the ASM model. The final phase will be used for completion of the thesis report and final testing of the software.

5.3 Project Schedule

5.3.1 Project Task Set

The project task set and phases are set out in table 4.3.1.

Timing	Task	Phase
20 Aug – 24 Aug	Project Plan	1
25 Aug – 31 Aug	Lit Review and Design	2
01 Sep – 14 Sep	ASM Prototype	3
15 Sep – 16 Oct	GUI Development	4
17 Oct – 22 Oct	Report and Testing	5

Table 5.3.1: Task Phases

5.3.2 Functional Decomposition

The system's functional composition is broken down into modules in Fig. 5.3.2

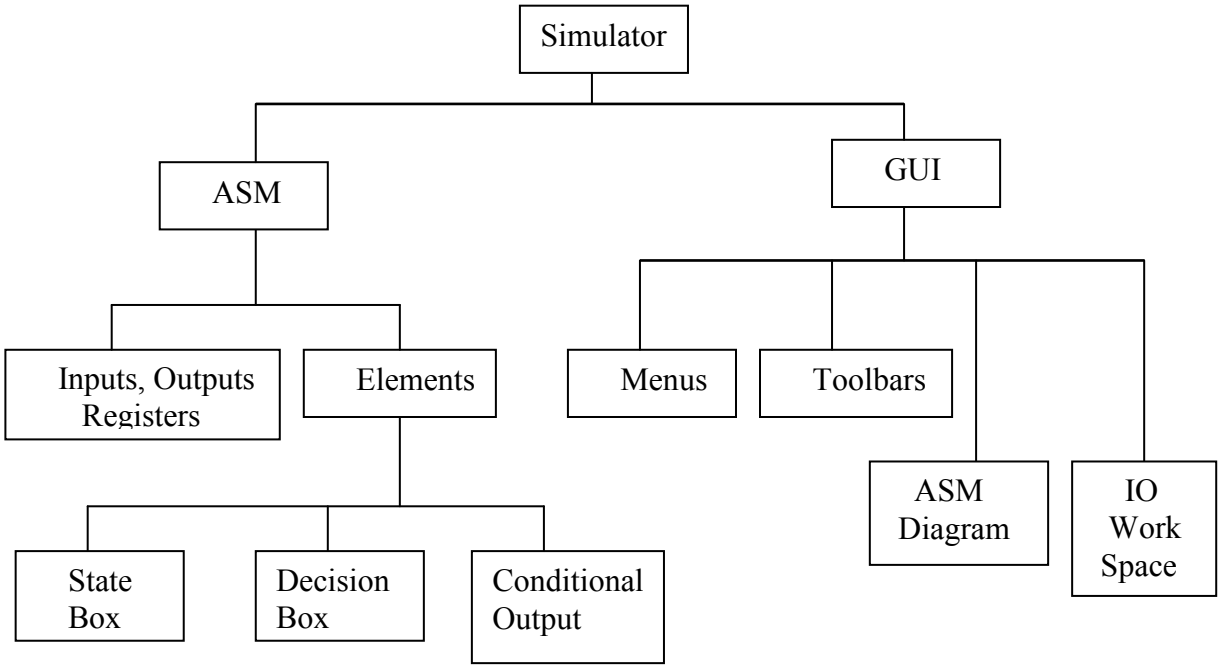


Fig. 5.3.2: System Decomposition

The functionality of each of the system modules is laid out in Table 5.3.3.

Module	**Functions**
Variable	
Input	Associate with decision box; Assert / deassert
Output/Register	Associate with State or Conditional Output
Element	Connect to other element
State Box	Update output / registers
Decision Box	Determine exit path
Conditional Output	Update output / registers
Menus	
Help menu	Display Program Help
File menu	New ASM; Save ASM; Load ASM; Close ASM
Tools menu	Verify machine; Run machine; Step Machine; Suspend Execution; Reset Machine
Toolbars	Add element to ASM space; Add variable to IO space; verify, run and step ASM;
ASM diagram	Add element
IO workspace	Add variable

Table 5.3.3: Module Functionality

CHAPTER 6

SOFTWARE DESIGN

The first step of the software design process consists of specifying the system use-case models. From these models, Class-Responsibility-Collaborator or CRC cards are derived. A UML class relationship model is then developed directly from these CRC cards. Following this, the system algorithm is modelled using object behaviour models (UML interaction and state diagrams). Finally a code skeleton can be drawn up, which is used to build a system prototype.

6.1 Use-Case Specifications

The various system use-cases are developed in this section. Each use-case specification includes:

- A use-case identifier.
- A brief description of the use case.
- A list of actors involved in the use case.
- The use-case diagram itself.
- A more detailed description of the basic flow of the use-case.
- Possible alternative flows of the use-case.
- Use-case pre-conditions and post-conditions.

6.1.1 Place Element Use-Case

Identifier

The “place element” use-case is assigned identifier UC1.

Brief Description

In this use-case the user wishes to place an element of a particular type on the ASM diagram. Once the type of element is selected, the user can place the element on the ASM diagram in a desired position.

Actors

The actors involved in this use case are the user, the element toolbar, and the element itself.

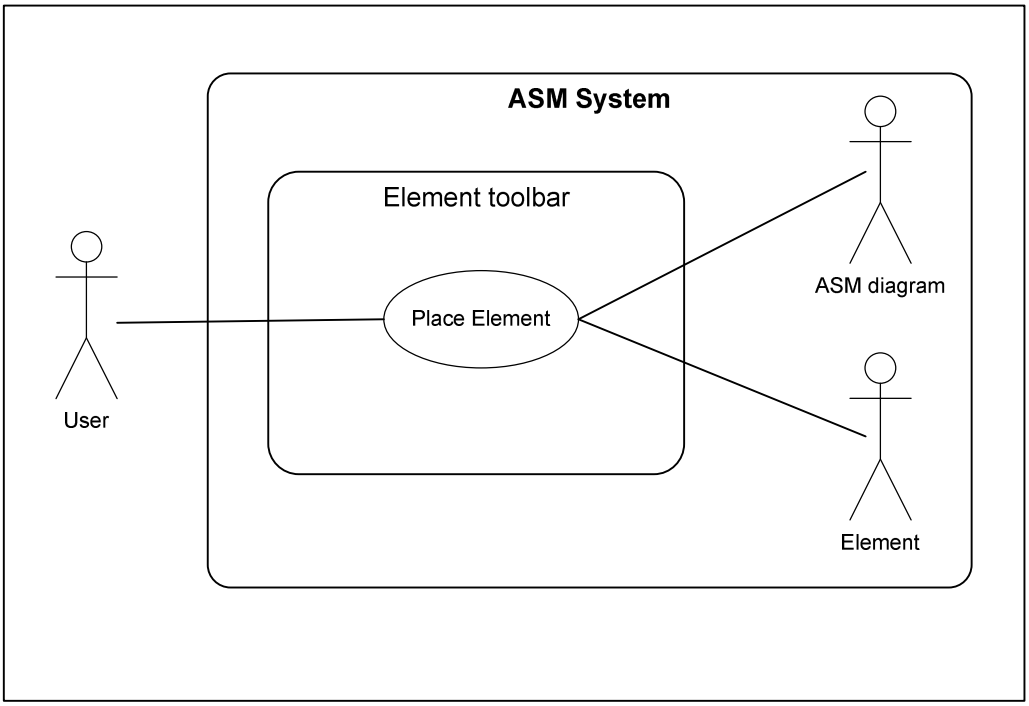


Fig. 6.1.1: Place Element Use-Case – UC1

Basic Flow

i. The user left clicks on an element icon in the element toolbar, changing the mouse to an element placement cursor and switching the system to placement mode.
ii. The user releases the mouse button, and moves the cursor onto the ASM diagram.
iii. The user clicks the left mouse button. If the cursor is positioned in an area of the diagram that does not infringe on any elements already part of the ASM, the new element is placed in that area.
iv. The system remains in placement mode allowing the user to place another element of the same type, or right click to return the system to idle mode, ending the use-case.

Alternative Flow - User Cancels Placement

i. The user left clicks on an element icon as before.
ii. The user presses the right mouse button before placing the element, canceling placement and returning the system to idle mode. This ends the use-case prematurely.

Alternative Flow - User Initiates another Use-case

i. The user left clicks on an element icon as before.
ii. The user initiates a new "place element" (UC1) use-case. This ends the in progress UC1 use-case prematurely. The use-case may also be suspended temporarily by an "add variable" (UC4) or "edit variable" (UC5) use-case. Once the UC4 or UC5 use-case completes the suspended "place element" use-case resumes.

Preconditions

The system must be in idle, connecting, select or placement mode when the use-case begins. If the user initiates a placement while a "connect elements" (UC3) or "place element" (UC1) use-case is currently underway, the in progress use-case is aborted.

Post-conditions

Once the use-case ends the system remains in placement mode until the user presses the right mouse button to enter idle mode.

6.1.2 Edit Element Parameters Use-Case

Identifier

The "edit element parameters" use-case is assigned identifier UC2.

Brief Description

The user wishes to edit one of the parameters of an element on the ASM diagram. The name of a state may be changed or variables from the active variable list may be associated with the element.

Actors

The actors involved in this use case are the user, the element being edited and the active variable list.

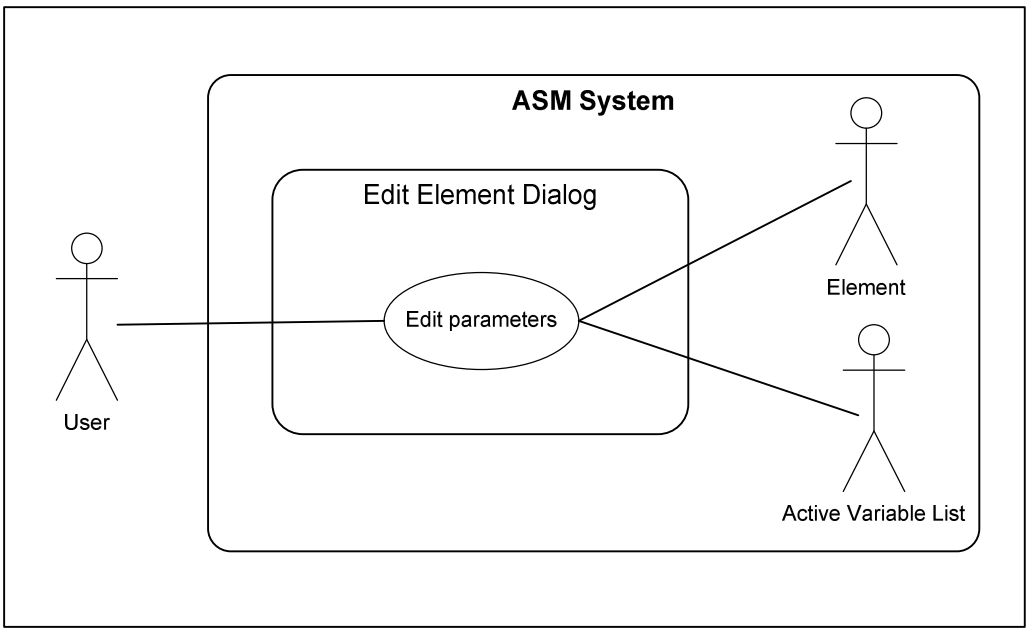


Fig. 6.1.2: Edit Element Parameters – UC2

Basic Flow

i. The user double clicks on an element on the ASM diagram with the left mouse button.

ii. An edit element parameters dialog box displays. This dialog shows a list of the variables which can be associated with this particular element. If the element selected is a state box the dialog contains a change name field.

iii. The user selects which variables to associate with the element, and gives the element a name if it is a state.

iv. If a register variable is chosen to associate with the element, the user specifies what value to move to the register when control passes to the element. If an invalid value is given the default value of zero is assigned.

v. The user presses the "ok" button ending the use-case.

Alternative Flow – Vector Decision Box

i. The user double clicks on a vector decision box.

ii. The dialog shown has a list of vector inputs currently active as well as their respective lengths. The user selects a vector input of length 'n' to associate with the vector decision box. If the vector decision box currently outgoing connections that exceed 2n in number, the user is warned that making the association will cause the decision box to lose all its outgoing connections. The user can then choose whether or not to associate the vector decision box with the selected input.

Preconditions

The system must be in idle or select mode to initiate an “edit element parameters” use-case.

Post-conditions

Once the use-case ends the system returns to idle mode.

6.1.3 Connect Elements Use-Case

Identifier

The “connect elements” use-case is assigned identifier UC3.

Brief Description

The goal of the use-case is to connect two ASM elements on the ASM diagram together. The user wishes to connect the exit path of one ASM element to the entry path of another. The entry path of a conditional output box can only be connected to the exit path of a decision box. If the user attempts to connect any other element to the entry of a conditional output box, the connection will not be allowed.

Actors

The actors involved in this use case are the user and the two elements being connected.

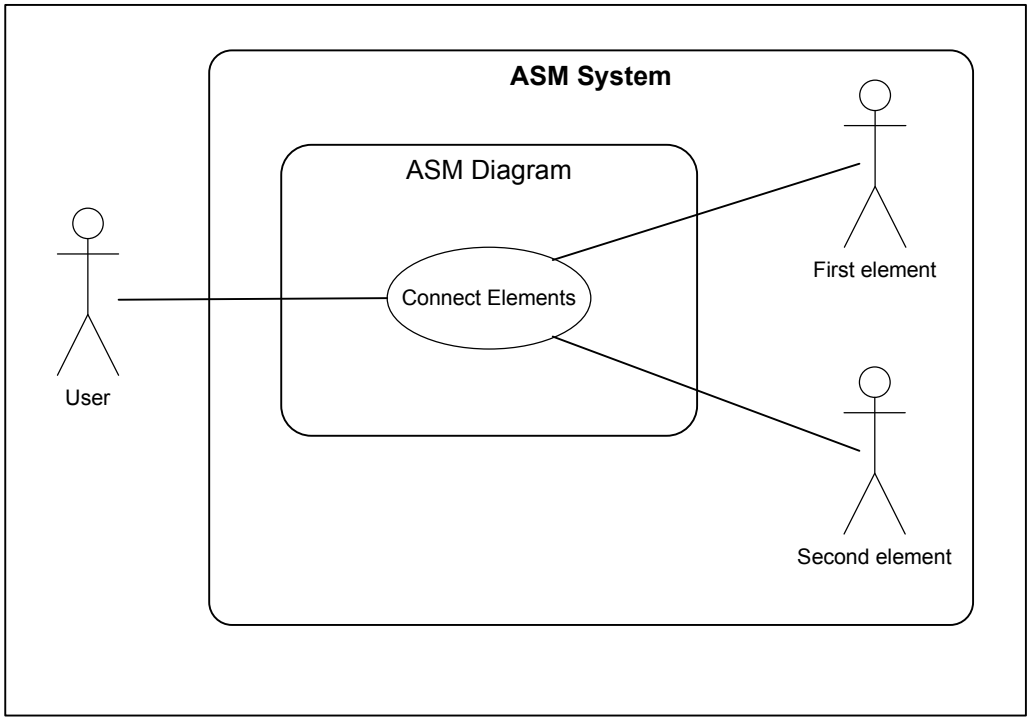


Fig. 6.1.3: Connect Elements Use-Case – UC3

Basic Flow

i. The user right clicks on an element on the ASM diagram, changing the mouse to a connection cursor and switching the system to connecting mode.

ii. The user can either click in empty space on the ASM diagram, or on another element.

   a. If the user left clicks on an empty area of the ASM diagram while in connecting mode, a connecting line is drawn from the last connecting point to the new point selected. Note that the first connecting line drawn from an element begins at the exit point of the originating element. After drawing the connecting line the system stays in connecting mode, the use-case flow of control returns to step (ii) above.

   b. If the user left clicks on a second element the system draws a connecting line to the new element and makes the connection if it is valid. A valid connection results in the system returning to idle mode and the use-case ends. An attempt to make an invalid connection results in the display of an error message, and the system remains in connection mode.

Alternative Flow – Connecting a Decision Box

i. The user right clicks on a vector decision box.

ii. If no vector input has yet been specified for the decision box, or if the input has no length specified yet, no connections may be made from the decision box. When the user connects the output of the vector decision box to another element, the user is prompted to choose the value of the input vector that will result in that decision path being chosen during the execution of the ASM.

Alternative Flow – Connection Already Exists

i. The user right clicks on an element as before, but this time the element already has a connection to another element.

ii. If the element is a vector decision box, the user can make up to 2^n connections from the element, where n is the length of the vector input associated with the decision box. If the element is a state or conditional output, or if the decision box has reached its maximum number of connections allowed, then the existing connection(s) is (are) deleted (after warning the user that making a new connection will do so). A new connection use-case then begins.

Alternative Flow - User Cancels Connection

i. The user right clicks on an element as before.

ii. The user may cancel the connection at any time by right clicking, returning the system to idle mode and deleting all connecting lines drawn during the use-case.

Preconditions

The system must be in idle mode to initiate a "connect element" use-case.

Post-conditions

Once the use-case ends the system returns to idle mode.

6.1.4 Select Element Use-Case

Identifier

The "select elements" use-case is assigned identifier UC4.

Brief Description

The goal of the use-case is to either remove an element from the ASM diagram, or move it to a new position. If the element is deleted, all connections associated with the element are lost on deletion. Once an element is selected, moving it is effectively the same as placing it (UC1).

Actors

The actors involved in this use case are the user, the element and the ASM diagram.

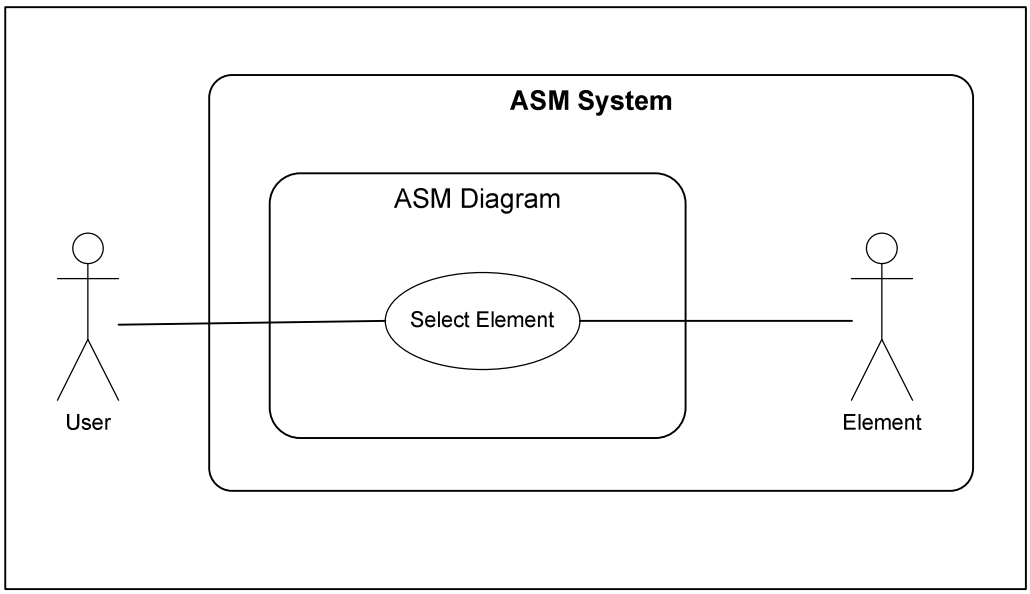


Fig. 6.1.4: Select Element Use-Case – UC4

Basic Flow

i. The user left clicks an element on the ASM diagram activating select mode. This highlights the selected element.
ii. The user has various options from this stage.
iii. If the user left clicks on empty space on the ASM diagram the element is moved to that position.
iv. If the user presses delete, the deletion must be confirmed, then the element is deleted permanently along with all of its connections and associations with active variables.
v. If the click the right mouse button the selection is cancelled.
vi. This ends the use case.

Alternative Flow – User Starts another Use-Case

i. The user selects an element on the ASM diagram.
ii. The user starts a new use case. The in progress use-case is either aborted or suspended depending on what type of use-case pre-empts it.
iii. If the user starts a "place element" (UC1), "edit element" (UC2), any ASM simulation use-case (UC8 to UC10) or file use-case (UC11 to UC13) the in progress use-case is aborted.
iv. If the user starts any other valid use case, the in progress use-case is suspended.

Preconditions

The system must be in idle mode to initiate a "delete element" use-case.

Post-conditions

Once the use-case ends the system returns to idle mode.

6.1.5 Add Variable Use-Case

Identifier

The "add variable" use-case is assigned identifier UC5.

Brief Description

In this use-case the user wishes to add a variable of a particular type from the variable toolbar to the active variable list. If the variable selected is a vector input, the user will be prompted to specify the length of the vector.

Actors

The actors for this use case are the user, the "add variable" dialog and the active variable list.

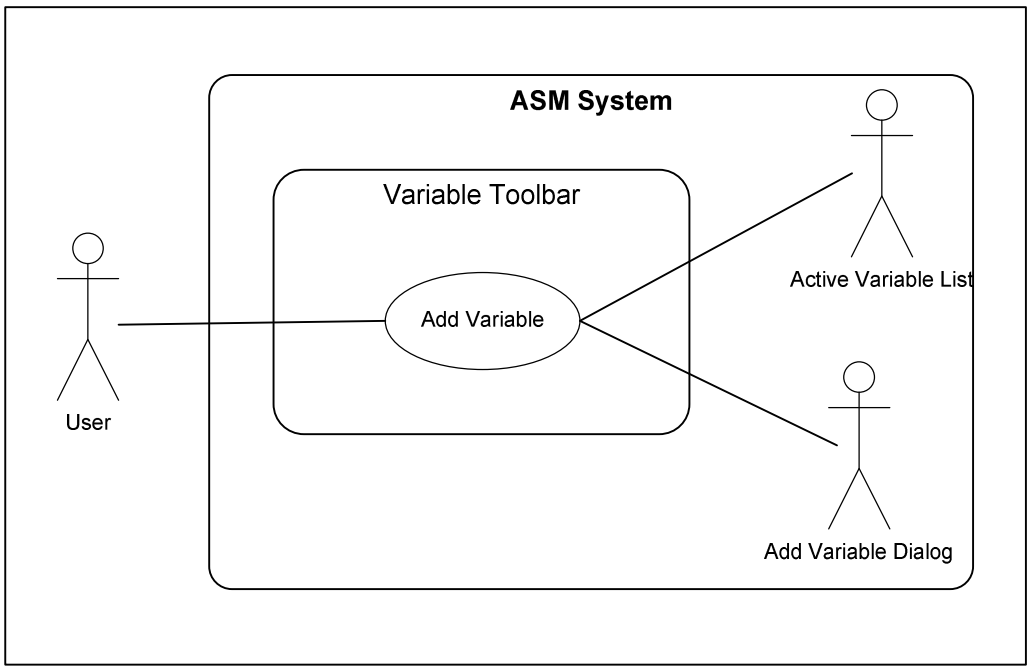


Fig. 6.1.5: Add Variable Use-Case – UC5

Basic Flow

i. The user left clicks on a variable type on the variable toolbar.

ii. The "add variable" dialog box opens. The user enters a unique name for the variable and if the variable is of type "input vector" the user also enters an integer length between 1 and 4 for the vector.

iii. The user presses enter or the "ok" button of the dialog. The active variable list is then updated with the new variable information and the use-case ends.

Alternative Flow – Invalid Variable Parameter Given

i. The user left clicks on a variable type and the "add variable" dialog opens as before.

ii. If the user does not enter a name for the variable, or the name is not unique, an error message displays and the user is prompted to re-enter the name. If the length of a vector input is invalid an error message is displayed and a default length of 1 is used.

Preconditions

The system may be in idle mode, connecting mode, select mode or placement mode to initiate an "add variable" use-case.

Post-conditions

Once the use-case ends the system returns to idle mode.

6.1.6 Edit Variable Use-Case

Identifier

The "edit variable" use-case is assigned identifier UC6.

Brief Description

The user's objective in this use case is to change one of a list of parameters in the active variable list. An appropriate "variable edit" dialog box will allow the user to make the desired changes. The user can change a variable name, stimulate an input variable, or change the value of a vector input.

Actors

The actors in this use case are the user and the active variable list.

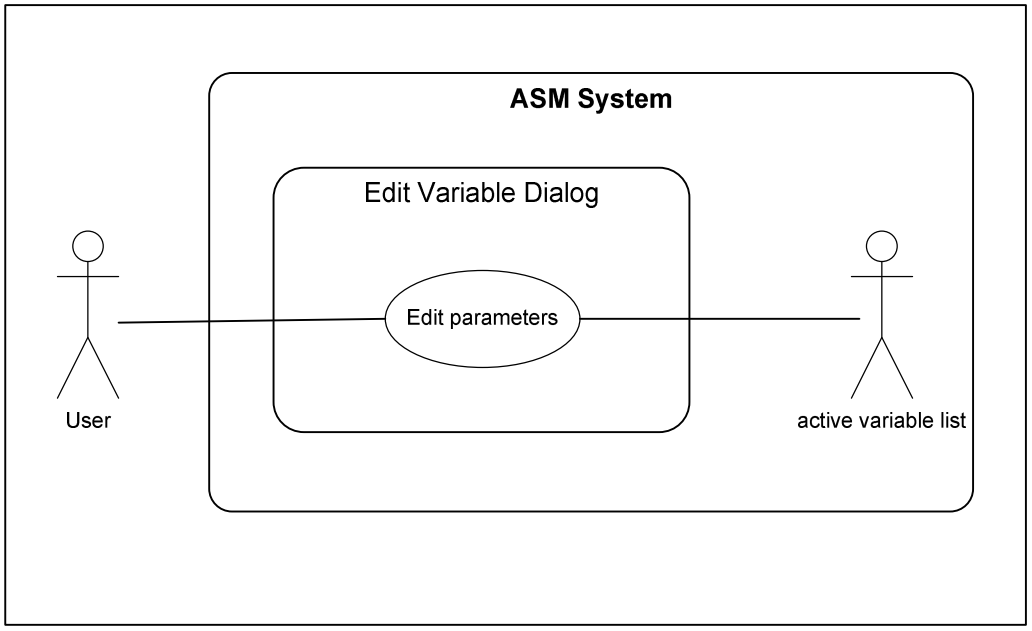


Fig. 6.1.6: Edit Variable Parameters Use-Case – UC6

Basic Flow – Change Name

i. The user left clicks on the "name" field of a particular variable in the active variable list.
ii. An "edit variable name" dialog displays. The user enters a new name for the variable and presses the ok button. NB if an invalid name is entered the user is prompted to re-enter (see alternative flow of UC4). This ends the use-case.

Basic Flow – Stimulate Variable

i. The user left clicks on the "stimulate variable" field of a particular variable in the active variable list.
ii. The variable value is inverted. This ends the use-case.

Basic Flow – Change Vector Value

i. The user left clicks on the "change value" field of an input vector variable in the active variable list.
ii. A "set new vector value" dialog is displayed allowing a new value for the vector to be entered. The value must be between 0 and 2^n where n is the length of the vector. If the user attempts to enter an invalid value an error message is displayed and the value does not change. This ends the use-case.

Preconditions

The system may be in idle mode, connecting mode, or placement mode to initiate an "Edit Variable" use-case.

Post-conditions

Once the use-case ends the system returns to the mode it was in prior to the initiation of the use-case.

6.1.7 Delete Variable Use-Case

Identifier

The “edit variable” use-case is assigned identifier UC7.

Brief Description

The user’s objective in this use case is to delete one of the variables in the active variable list. The variable is removed from the list of associated variables of any elements that it has been associated with. If the variable is a vector input associated with a vector decision box, then that decision box loses its connections.

Actors

The actors in this use case are the user and the active variable list.

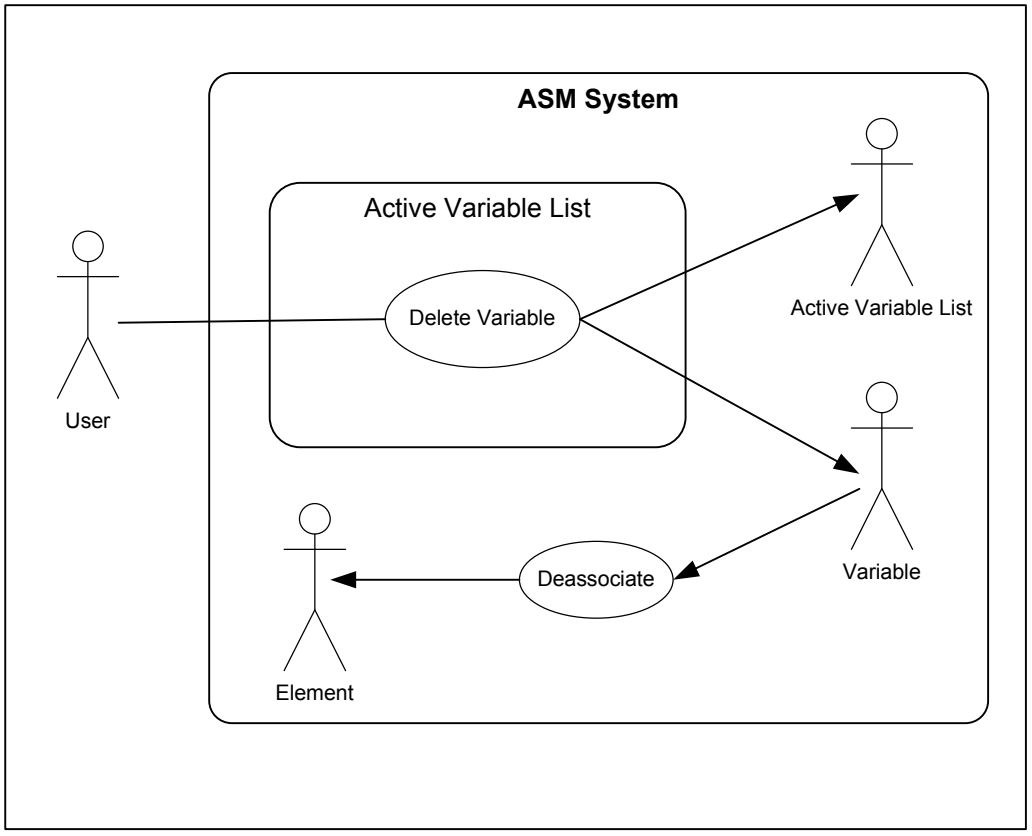


Fig. 6.1.7: Delete Variable Use-Case – UC7

Basic Flow

i. The user right clicks on a variable entry in the active variable list.
ii. The user is prompted to confirm that the variable will be deleted. The user may choose “yes” or “no”
iii. If the user chooses “yes” the variable is removed from the variable list, and all associations with elements are deleted. If the variable was associated with any vector decision boxes, those boxes lose their connections.
iv. If the user chooses “no” the use-case is aborted.
v. This ends the use case.

Preconditions

The system may be in idle mode, connecting mode, placement mode or select mode to initiate this use-case.

Post-conditions

Once the use-case ends the system returns to the mode it was in prior to the initiation of the use-case.

6.1.8 Run Machine Use-Case

Identifier

The “run machine” use-case is assigned identifier UC8.

Brief Description

The goal of the use-case is to simulate the running of the ASM constructed on the diagram. When the machine is simulated by the ASM system, an internal clock is automatically generated by the system. The user may stimulate inputs and observe outputs as the machine runs.

Actors

The actors of this use case are the user and the ASM simulator.

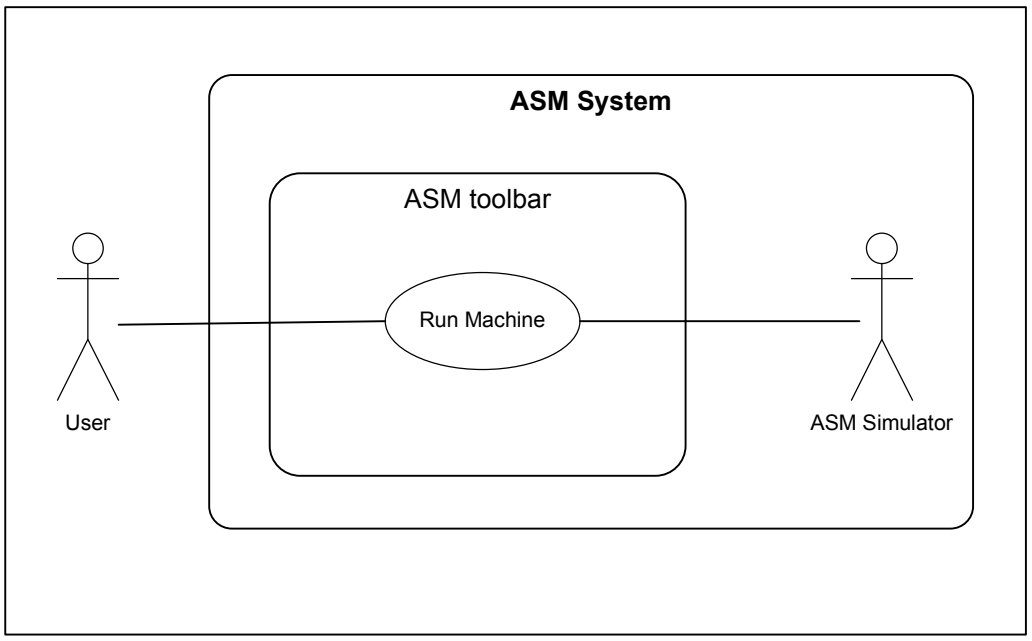


Fig. 6.1.8: Run Machine Use-Case – UC8

Basic Flow

i. The user left clicks on either the "run machine" icon of the ASM toolbar or the equivalent option in the machine menu.
ii. The system simulates the ASM using the run machine algorithm, clocked by the internal system clock. The user may direct the flow of the machine by stimulating inputs and may observe outputs changing on the active variable list.
iii. The user stops the simulation using the "stop machine" icon on the ASM toolbar or the equivalent option on the machine menu. This ends the use-case.

Alternative Flow – Elements Not Connected

i. The user selects "run machine" from the toolbar or machine menu.
ii. The system detects that some elements have unconnected exit or entry paths and warns the user.
iii. The user may either abort the simulation and end the use-case or run the partially connected machine as is.

Alternative Flow – Initial State Not Specified

i. The user selects "run machine" from the toolbar or machine menu.
ii. The system recognizes that the initial state has not been specified, displays an appropriate error message and aborts the simulation.
iii. The use-case ends.

Preconditions

The system must be in idle or suspended mode to initiate the “run machine” use-case.

Post-conditions

Once the use-case ends the system is in suspended mode.

6.1.9 Step Machine Use-Case

Identifier

The “step machine” use-case is assigned identifier UC9.

Brief Description

The goal of the use-case is to step a suspended ASM simulation forward one clock cycle. A simulation can be suspended by stopping a running ASM (UC8) or by a “step machine” (UC9) use-case ending.

Actors

The actors in UC9 are the user and the ASM simulator.

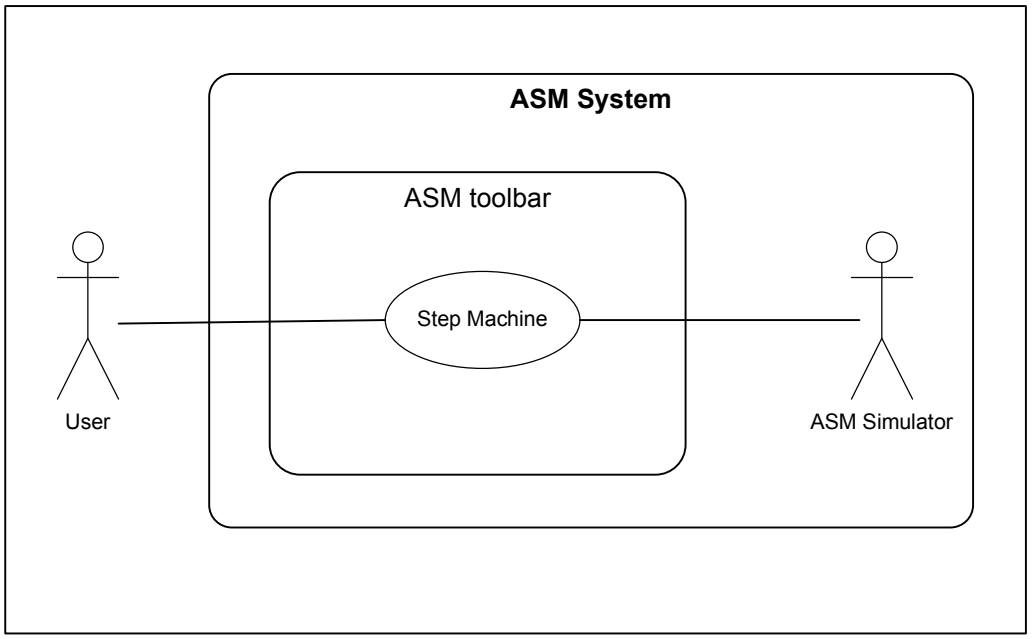


Fig. 6.1.9: Step Machine Use-Case – UC9

Basic Flow

i. The user left clicks on either the “step machine” icon of the ASM toolbar or the equivalent option in the machine menu.
ii. The system simulator moves the execution of the ASM forward one step using the run machine algorithm, effectively clocking the system once.
iii. Once the data-path routines have been processed for that clock cycle the simulator enters the suspended state. This ends the use-case.

Preconditions

The system must be in idle or suspended mode to initiate the “run machine” use-case.

Post-conditions

Once the use-case ends the system is in suspended mode.

6.1.10 Reset Machine Use-Case

Identifier

The “reset machine” use-case is assigned identifier UC10.

Brief Description

In this use case the user resets an ASM simulation which is either running currently or whose execution has been stopped or suspended. Resetting the ASM returns the system to idle mode.

Actors

The actors in UC10 are the user and the ASM simulator.

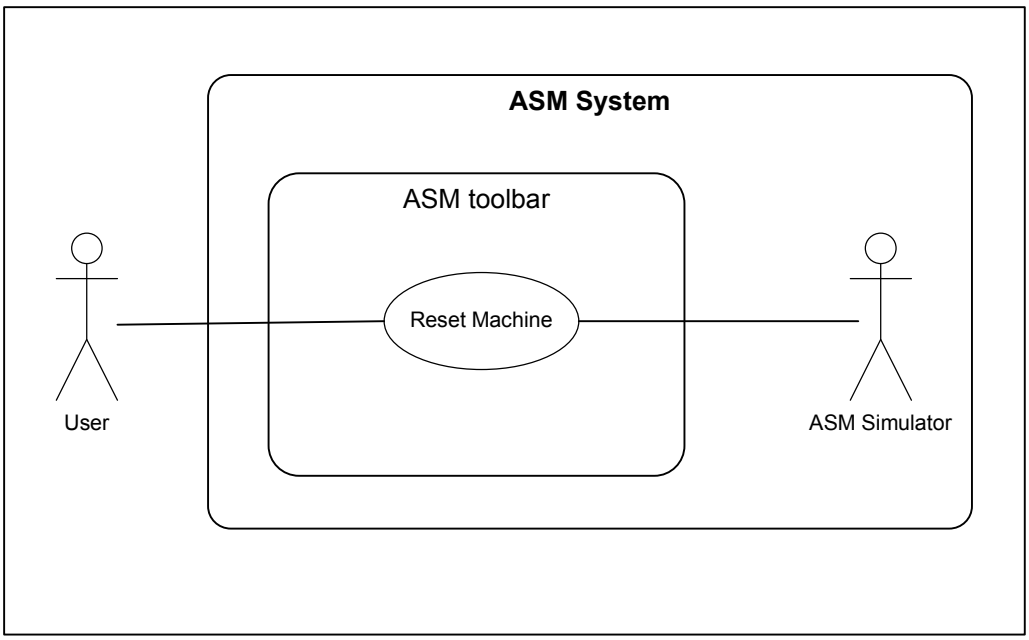


Fig. 6.1.10: Reset Machine Use-Case – UC10

Basic Flow

i. The user left clicks on either the "reset machine" icon of the ASM toolbar or the equivalent option in the machine menu.
ii. The system simulator shuts down, returning the ASM system to idle mode. This ends the use case.

Preconditions

The system must be in running or suspended mode to initiate the "reset machine" use-case.

Post-conditions

Once the use-case ends the system is in idle mode.

6.1.11 New ASM Use-Case

Identifier

The "new ASM" use-case is assigned identifier UC11.

Brief Description

In this use case the ASM diagram and active variable list are cleared, allocated memory is freed up and a new ASM diagram workspace is created.

Actors

The actors involved are the user, the active variable list, and the ASM diagram.

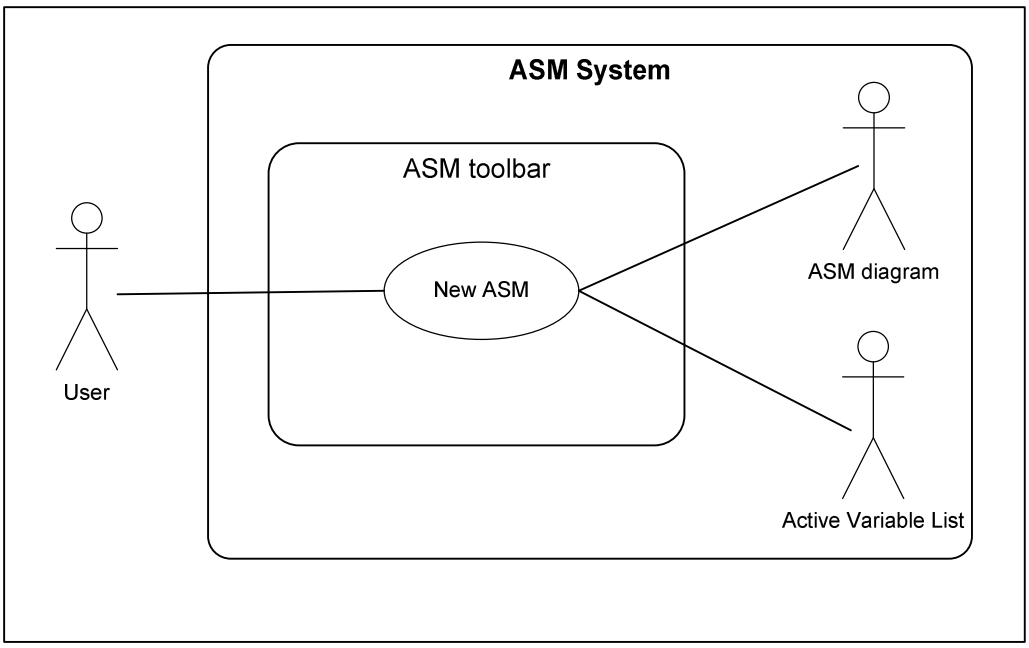


Fig. 6.1.11: New ASM Diagram Use-Case

Basic Flow

i. The user left clicks on the new ASM diagram icon on the ASM toolbar or selects the equivalent option from the file menu.
ii. If the current ASM diagram has not been saved in permanent storage the user is warned that creating a new ASM will destroy the current ASM.
iii. The user decides to save the current ASM, cancel the new ASM operation, or abandon the current ASM.
iv. If the user opts to save or abandon the current ASM, a new ASM workspace is created with a blank ASM diagram, active variable list and newly initialised local variables.

Preconditions

The system must not be in the running state to create a new ASM.

Post-conditions

Once the use-case ends the system is in the idle state.

6.1.12 Save ASM Use-Case

Identifier

The “save ASM” use-case is assigned identifier UC12.

Brief Description

The goal of UC12 is to save the current ASM in permanent memory. The file storage system captures the data associated with the current ASM diagram and active variable list and stores it in a file in permanent storage for later retrieval.

Actors

The actors of UC12 are the user, file system, ASM diagram and active variable list.

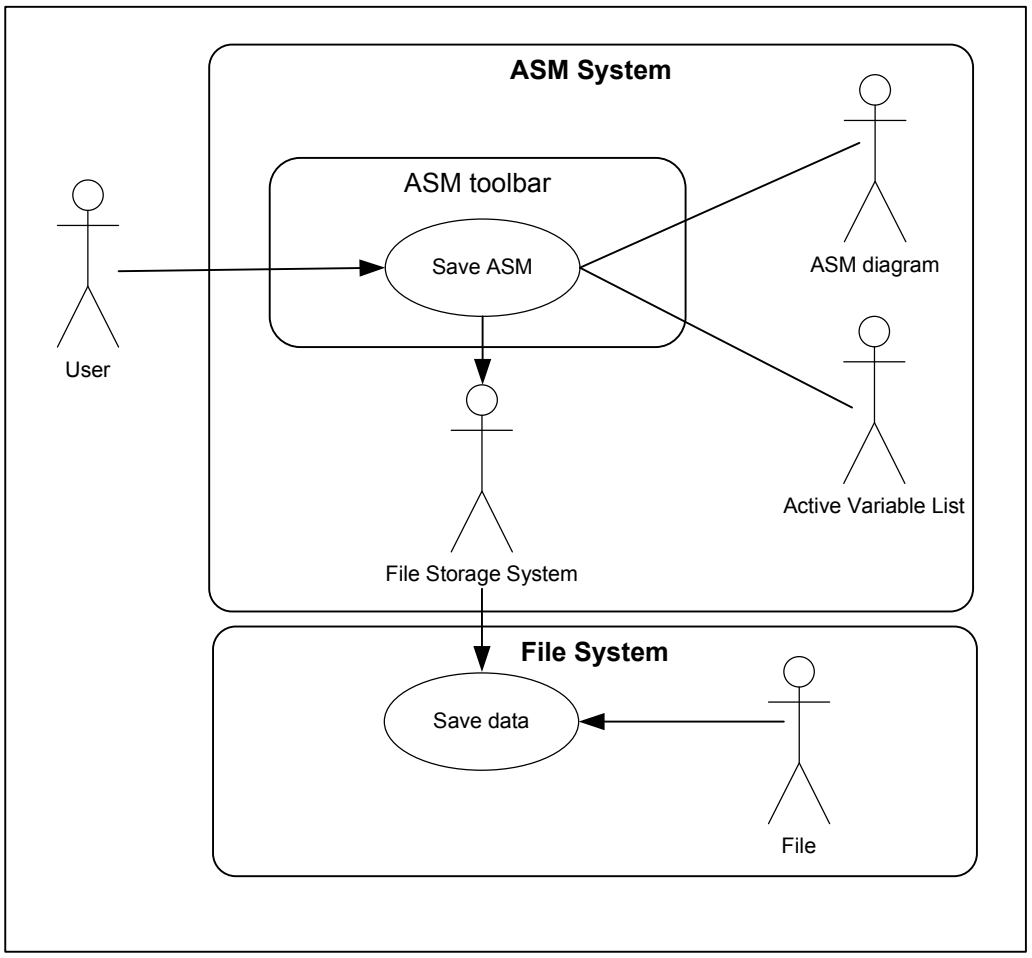


Fig. 6.1.12: Save ASM Diagram Use-Case – UC12

Basic Flow

i. The user left clicks on the save ASM diagram icon on the ASM toolbar or selects the equivalent option from the file menu.
ii. The user is then prompted in a "file save" dialog to provide a file name and directory location to save the ASM save file.
iii. The user may overwrite a previously saved file if desired but must first give confirmation that it is the intended action.
iv. Once the user has provided a file name and pressed the "ok" button the file storage system saves the data associated with the ASM diagram to file and ends the use case.

Alternative Flow – user cancels save

i. The user selects the save ASM diagram icon as before.
ii. Whilst within the "file save" dialog the user cancels the save aborting the use-case.

Alternative Flow – user does not enter filename

i. The user selects the save ASM diagram icon as before.
ii. The user attempts to save the file without providing a file name, resulting in an error message being shown and the save dialog re-displaying. The use case will end only when the user supplies a valid filename or cancels the use case.

Preconditions

The system must be in idle, connecting or placement mode to save the ASM to file. However, if the system is in connecting or placement mode, the connection or placement in progress is suspended and will not be saved to file.

Post-conditions

Once the use-case ends the system returns to the mode it was previously in.

6.1.13 Open ASM Use-Case

Identifier

The “open ASM” use-case is assigned identifier UC13.

Brief Description

In this use case the user wishes to retrieve a previously saved ASM diagram with its associated active variable list from file. The local variables, elements and connections of the ASM diagram as they were saved in permanent storage are all retrieved by the file retrieval system.

Actors

The actors of UC13 are the user, ASM diagram, active variable list and file system.

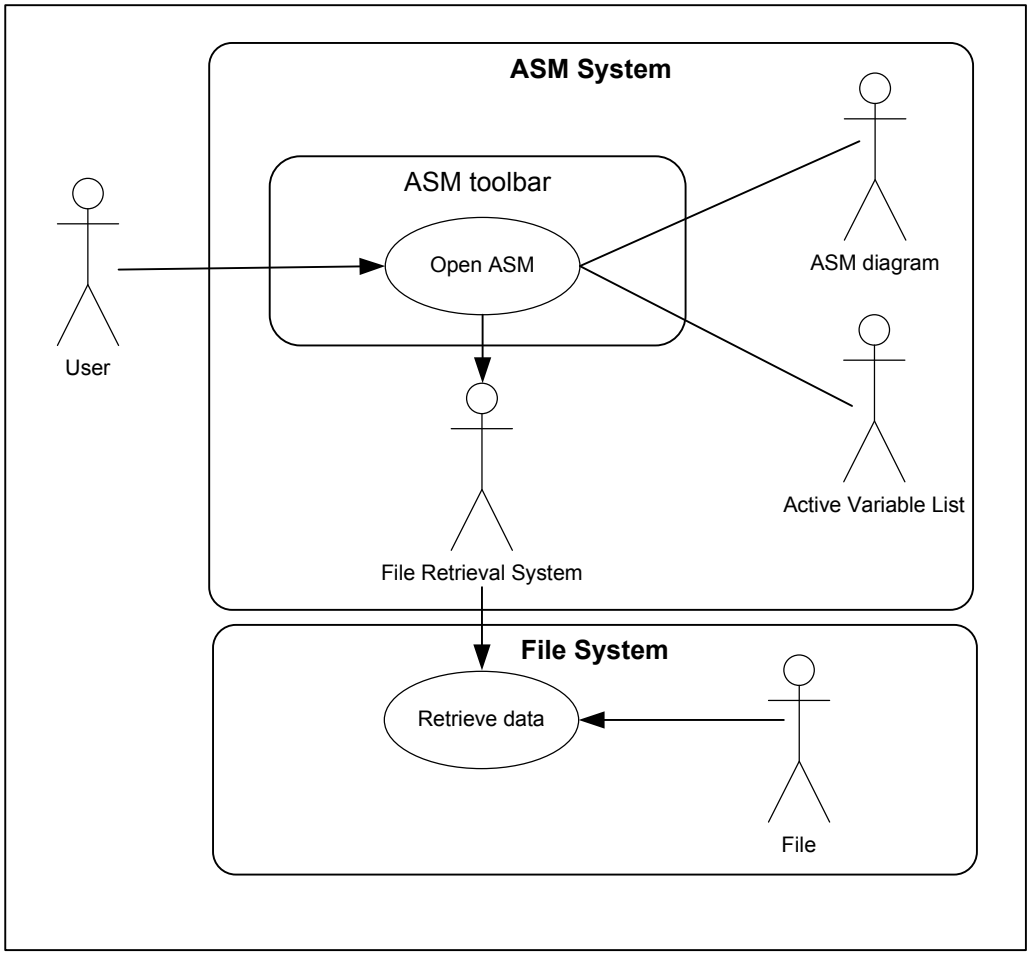


Fig. 6.1.13: Open ASM Diagram Use-Case – UC13

Basic Flow

i. The user left clicks on the open ASM diagram icon on the ASM toolbar or selects the equivalent option from the file menu.
ii. The user is then prompted in a "file open" dialog to select the ASM file to open.
iii. Once the user has selected a valid file and pressed the "ok" button the file retrieval system recovers the data associated with the ASM diagram from file. The ASM diagram and active variable data is entered into the current ASM workspace and all local variables from the retrieved ASM workspace are restored. This ends the use-case

Preconditions

The system may be in idle, connecting or placement mode when the use-case begins. However, if the system is in connecting or placement mode, the connection or placement is abandoned when the new ASM workspace is opened.

Post-conditions

Once the use-case ends the new ASM system is in idle mode.

6.1.14 Help Use-Case

Identifier

The "Help" use-case is assigned identifier UC14.

Brief Description

The goal of UC14 is to view help information concerning how to use the program. The user is given a choice to view either a dialog box giving basic instructions on how the program works or an about dialog displaying information program version, when it was written and so forth.

Actors

The actors involved in UC14 are the user and the help dialog.

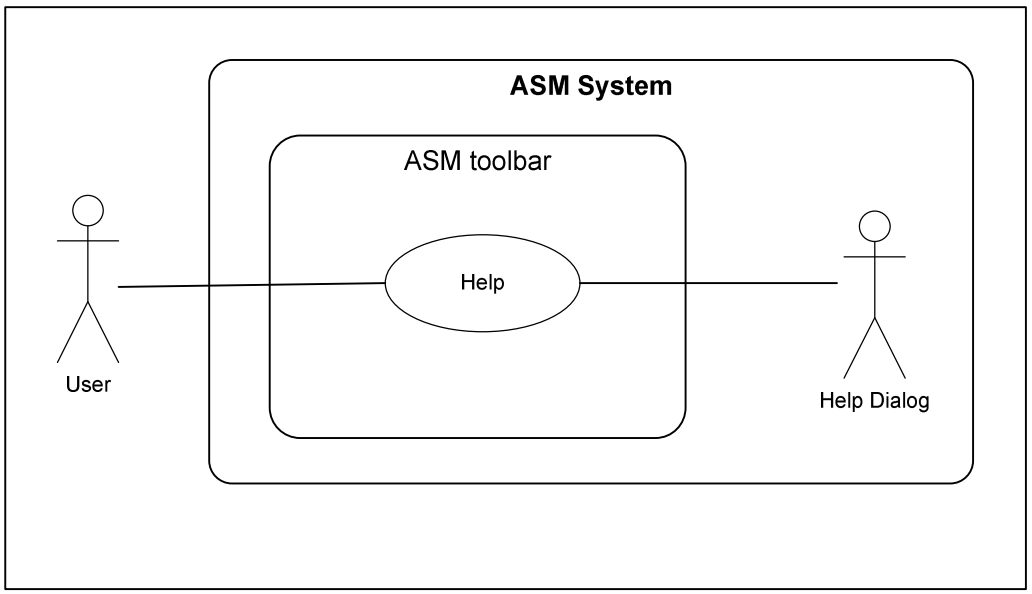


Fig. 6.1.14: Help Use-Case – UC14

Basic Flow

i. The user left clicks the help / about icon on the ASM toolbar or the equivalent help menu item.
ii. The help or about dialog is displayed.
iii. The user presses the "ok" or close button ending the use-case.

Preconditions

The system must be in idle, connecting or placement mode to enter this use case.

Post-conditions

Once the use-case ends the system returns to the mode it was in previously.

6.1.15 Set ASM Properties Use-Case

Identifier

The "set ASM properties" use-case is assigned identifier UC15.

Brief Description

The goal of this use-case is to set the initial state and clock speed of the ASM. When the user specifies the initial state the system does no checks to verify that the choice of state makes sense, but some indication of whether the ASM states are connected logically is given at run time (see UC8 "run machine"). The user also chooses from a range of clock speeds available for clocking the execution of the ASM simulator. If no clock speed has been specified at run time, a default is used.

Actors

The actors involved in the use-case are the user and the ASM properties dialog.

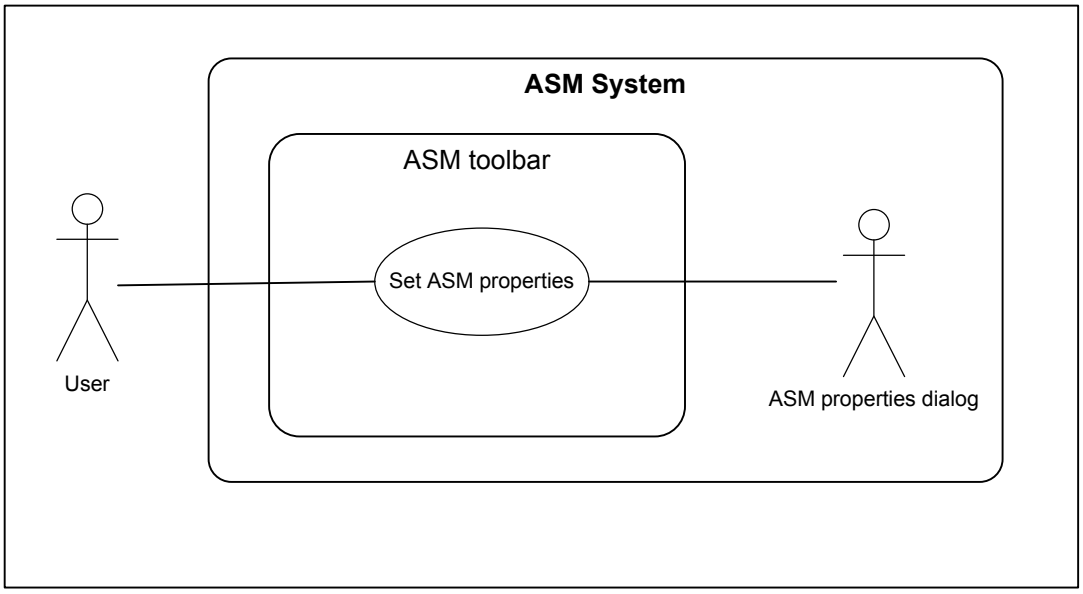


Fig. 6.1.15: Set ASM Properties Use-Case – UC15

Basic Flow

i. The user left clicks the "set ASM properties" icon on the ASM toolbar or selects the equivalent machine menu option.
ii. A dialog showing a list of the states currently on the ASM diagram and a clock speed spin control is shown for selection.
iii. The user selects a state and clock speed then clicks the "ok" button.
iv. The initial state and clock speed parameters of the system are saved. This ends the use case.

Alternative Flow – user cancels use-case

i. The user selects the "set ASM properties state" icon as before.
ii. From the "set ASM properties" dialog the user left clicks "cancel" aborting the use-case.

Alternative Flow – user does not select state

i. The user selects the "set ASM properties state" icon as before.
ii. From the "set ASM properties" dialog the user left clicks "ok" without selecting an initial state. The use case is aborted and no state is set as the initial.

Preconditions

The system may be in idle, connecting or placement mode to enter this use case.

Post-conditions

Once the use-case ends the system returns to its previous mode.

6.2 Class-Responsibility-Collaborator Cards

The CRC cards derived directly from the use-case specifications for the ASM simulator system are shown in Tables 6.2a to 6.2h that follow.

Class Name: ***ASM Simulator*** Class Type: *Device (simulator)* Class Characteristics: *tangible, aggregate, sequential, transient*	
Responsibilities	**Collaborators**
Attributes diagram ***Methods*** run simulation ( ) step simulation ( ) reset simulation ( ) stop simulation ( )	ASM Diagram ASM Toolbar System Clock

Table 6.2a: ASM Simulator CRC Card

Class Name: ***ASM Diagram*** Class Type: *Object (diagram)* Class Characteristics: *tangible, aggregate, concurrent, temporary*	
Responsibilities	**Collaborators**
Attributes elements array first element current element connections array ***Methods*** place element ( ) delete element ( ) connect element ( ) edit element ( )	ASM Dialogs ASM Toolbar Element ASM Simulator

Table 6.2b: ASM Diagram CRC card

Class Name: ***ASM Toolbar*** Class Type: *Interaction* Class Characteristics: *abstract, atomic, concurrent, temporary*	
Responsibilities	**Collaborators**
Attributes element select buttons file toolbar buttons machine toolbar buttons add variable buttons ***Methods*** button pressed ( )	ASM Simulator ASM Dialogs ASM Diagram Active Variable List

Table 6.2c: ASM Toolbar CRC card

Class Name: ***Active Variables List*** Class Type: *Interaction* Class Characteristics: *abstract, aggregate, concurrent, temporary*	
Responsibilities	**Collaborators**
Attributes variableList ***Methods*** add variable ( ) edit variable ( ) delete variable ( )	Variable ASM Dialogs

Table 6.2d: Active Variables List CRC card

Class Name: ***ASM Dialogs*** Class Type: *Interaction* Class Characteristics: *abstract, atomic, sequential, transient*	
Responsibilities	**Collaborators**
Attributes file open and save dialogs add element dialog add variable dialog edit element dialog edit variable dialog error and warning dialogs about and help dialogs ***Methods*** show dialog	File System ASM Diagram ASM Toolbar Active Variable List

Table 6.2e: ASM Dialogs CRC card

Class Name: *Element* Class Type: *Object* Class Characteristics: *tangible, atomic, sequential, transient*	
Responsibilities	**Collaborators**
Attributes type variablesArray next element vector length (if vector) ***Methods*** connect to next	ASM Diagram Active Variables List Variable

Table 6.2f: Element CRC card

Class Name: *Variable* Class Type: *Object* Class Characteristics: *tangible, atomic, sequential, transient*	
Responsibilities	**Collaborators**
Attributes name type value length	Active Variable List Element

Table 6.2g: Variable CRC card

Class Name: ***File System*** Class Type: *External entity* Class Characteristics: *abstract, atomic, sequential, transient*	
Responsibilities	**Collaborators**
Attributes File Retrieval System File Storage System ***Methods*** save ASM open ASM	ASM Dialogs

Table 6.2h: File System CRC card

6.3 Class Relationship Model

Fig. 6.3 shows the class relationship model for the ASM system.

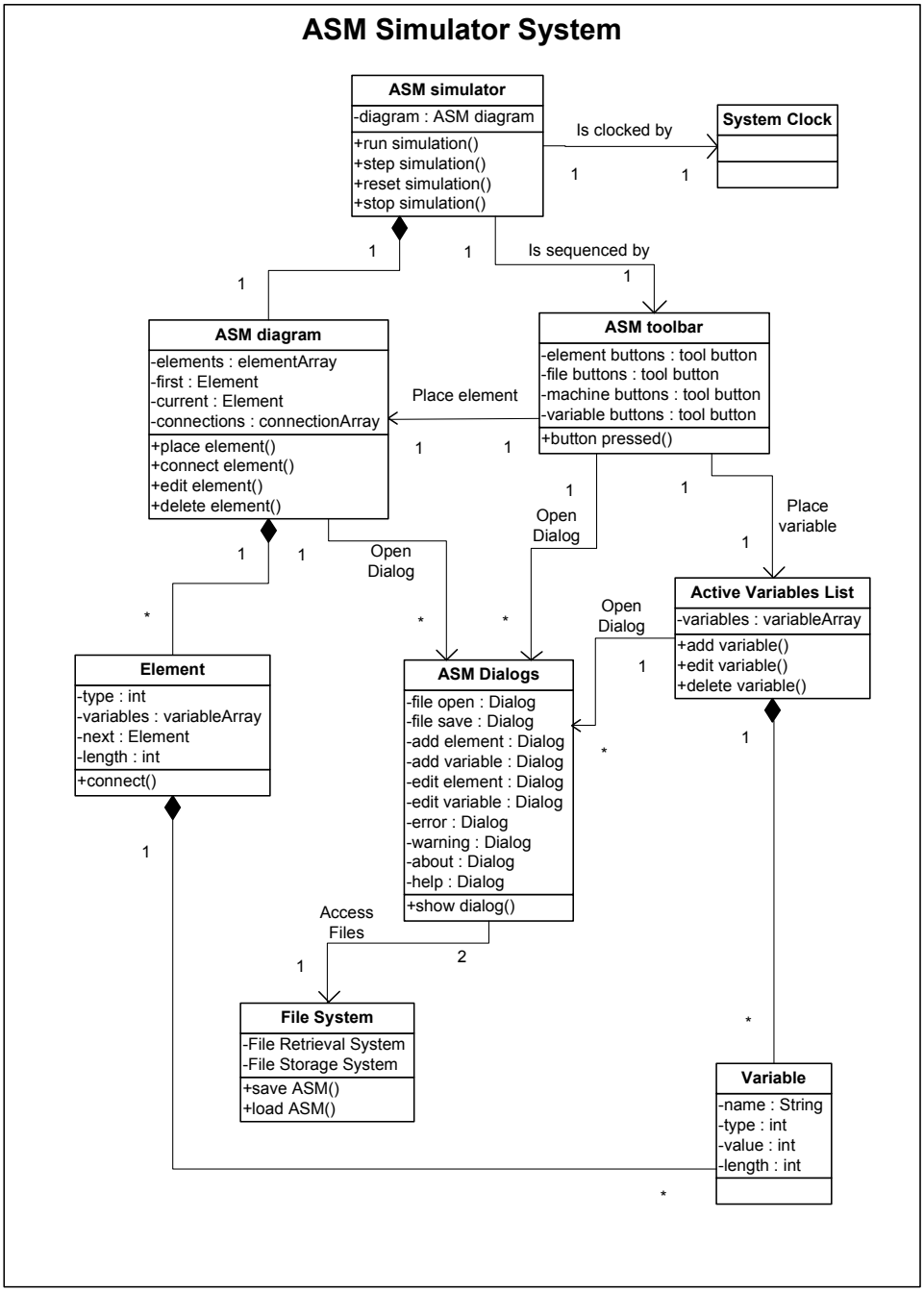


Fig. 6.3: ASM Simulator Object Relationship Model

6.4 Object Behaviour Models

The object behaviour models are used to specify the dynamic behaviour of the system. The two types of models used are sequence diagrams and UML state chart diagrams.

6.4.1 Sequence Diagram

The system sequence diagrams are derived directly from the use-case scenarios. The sequence diagrams are shown in Fig. 6.4.1a to 6.4.1h.

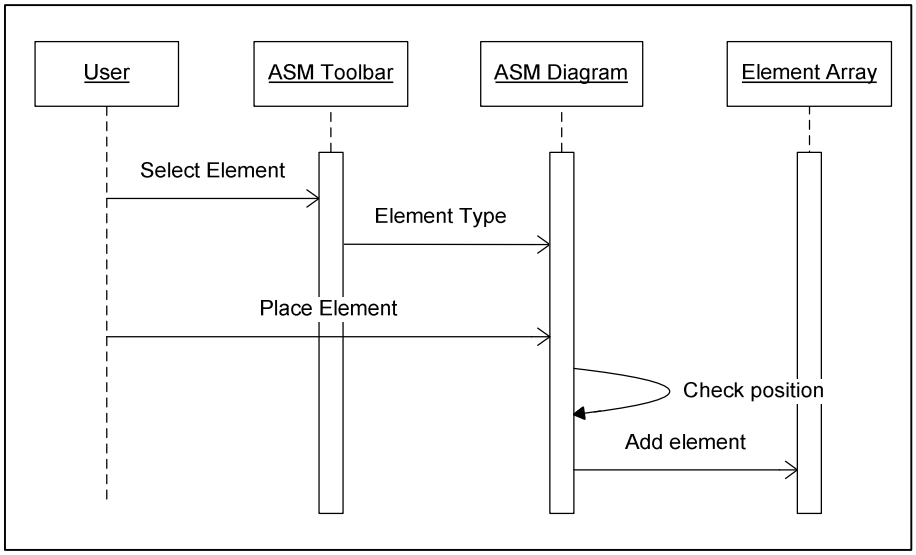


Fig. 6.4.1a: Place Element Sequence Diagram

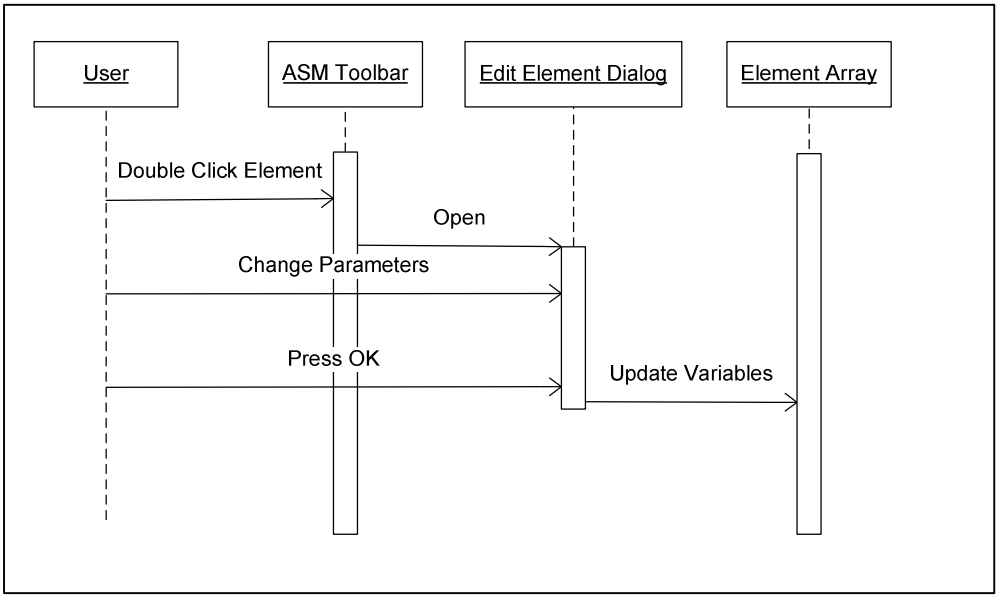


Fig. 6.4.1b: Edit Element Sequence Diagram

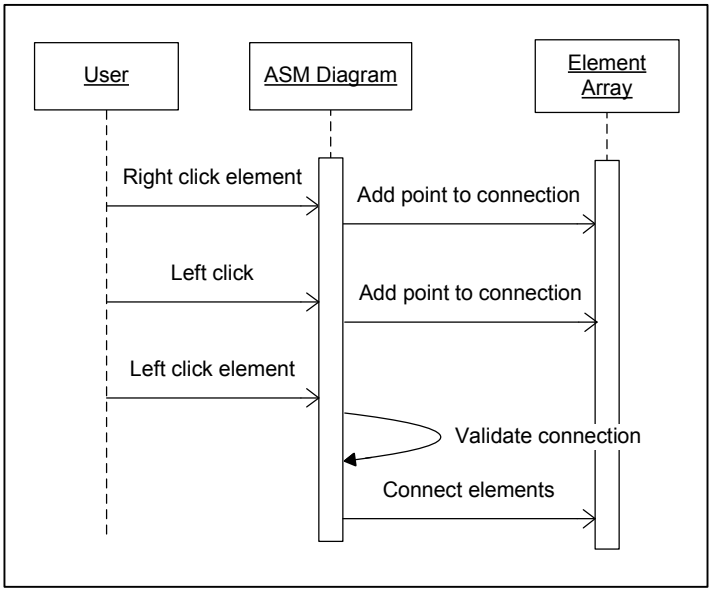


Fig. 6.4.1c: Connect Element Sequence Diagram

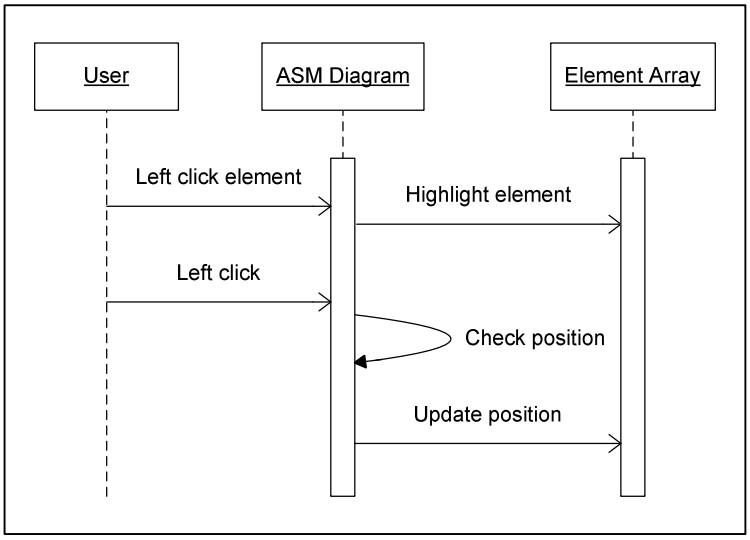


Fig. 6.4.1d: Move Element Sequence Diagram

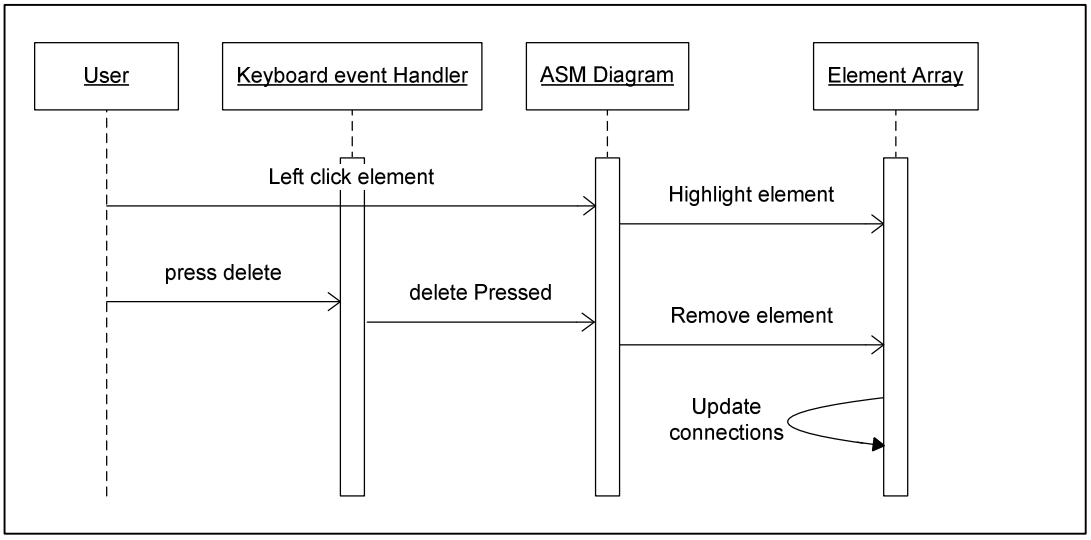


Fig. 6.4.1e: Delete Element Sequence Diagram

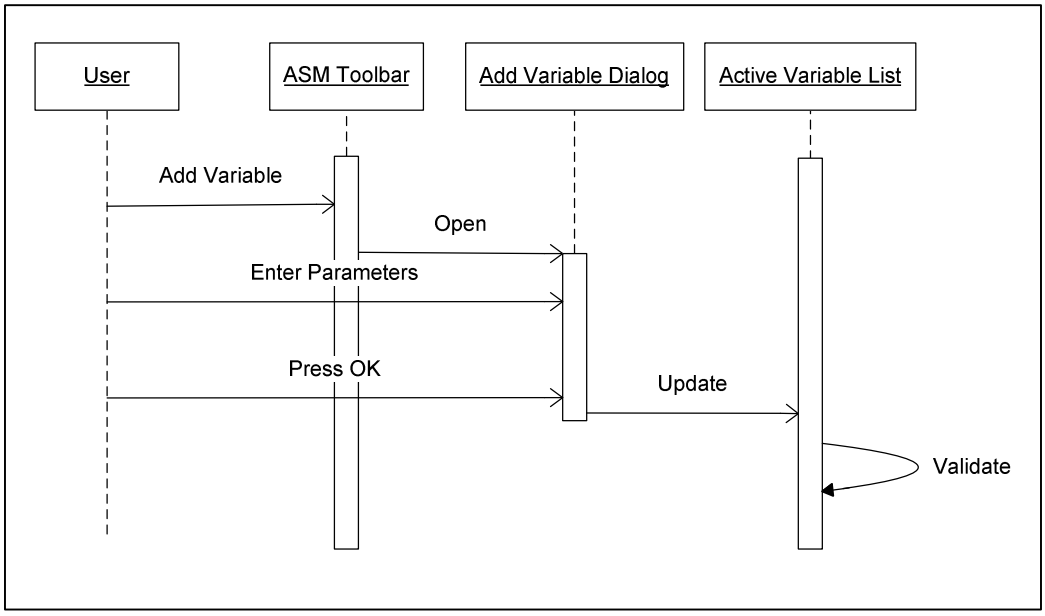


Fig. 6.4.1f: Add Variable Sequence Diagram

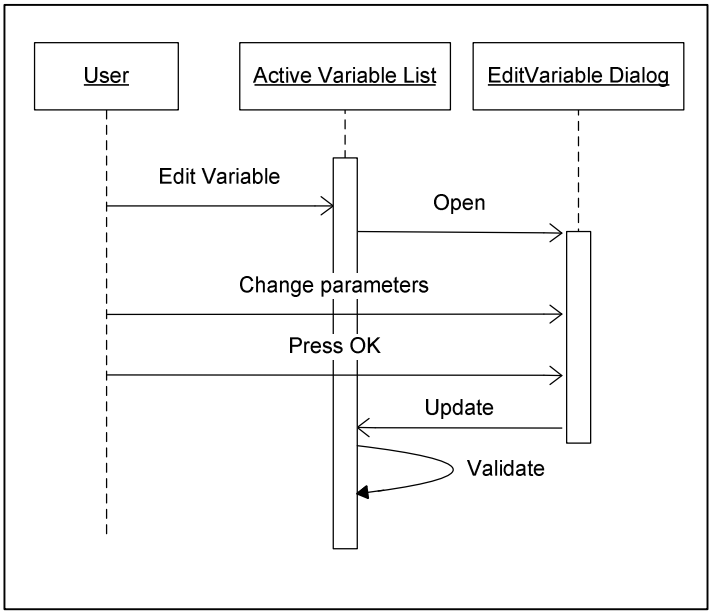


Fig. 6.4.1g: Edit Variable Parameters Sequence Diagram

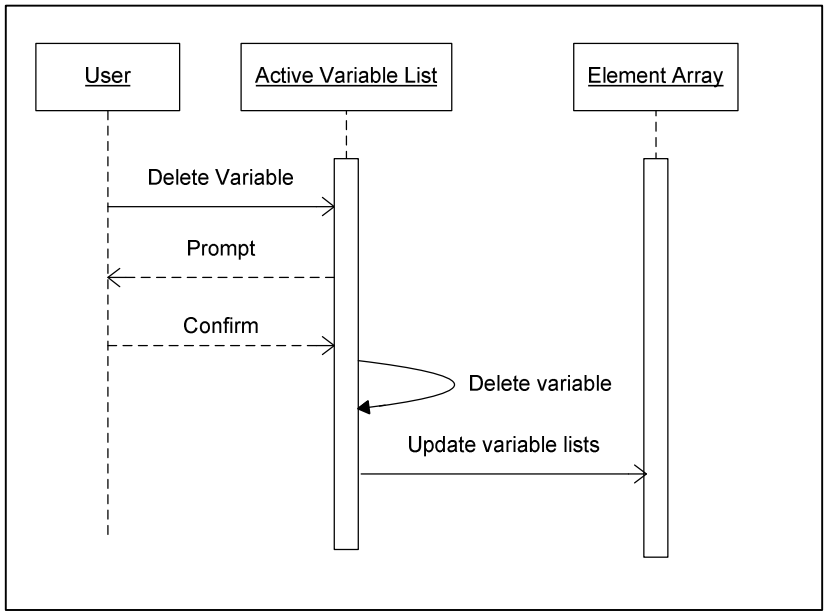


Fig. 6.4.1h: Delete Variable Sequence Diagram

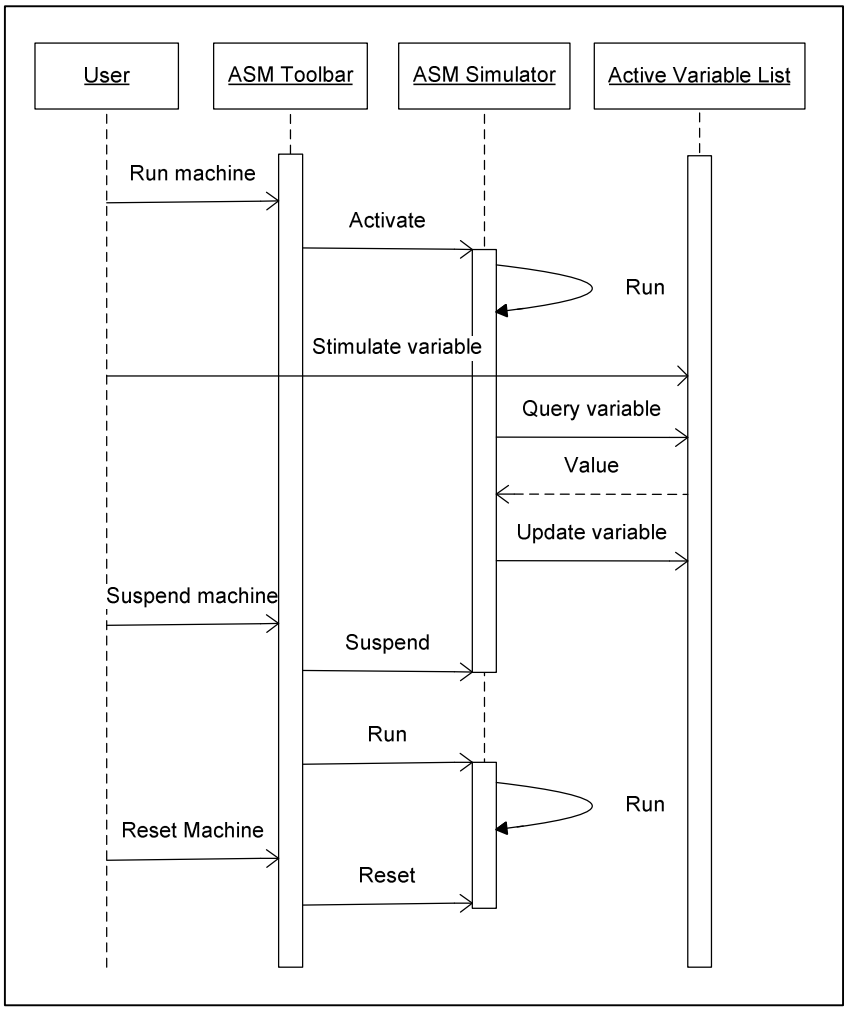


Fig. 6.4.1i: Simulate Machine Sequence Diagram

The remaining file system use-cases are relatively trivial in their sequencing and as such do not require sequence diagrams.

6.4.2 State Chart Diagram

The UML state chart modelling the ASM simulation sequence is shown in Fig. 6.4.2. The simulation sequence is effectively the same whether clocked by internal events or by the user manually stepping the machine.

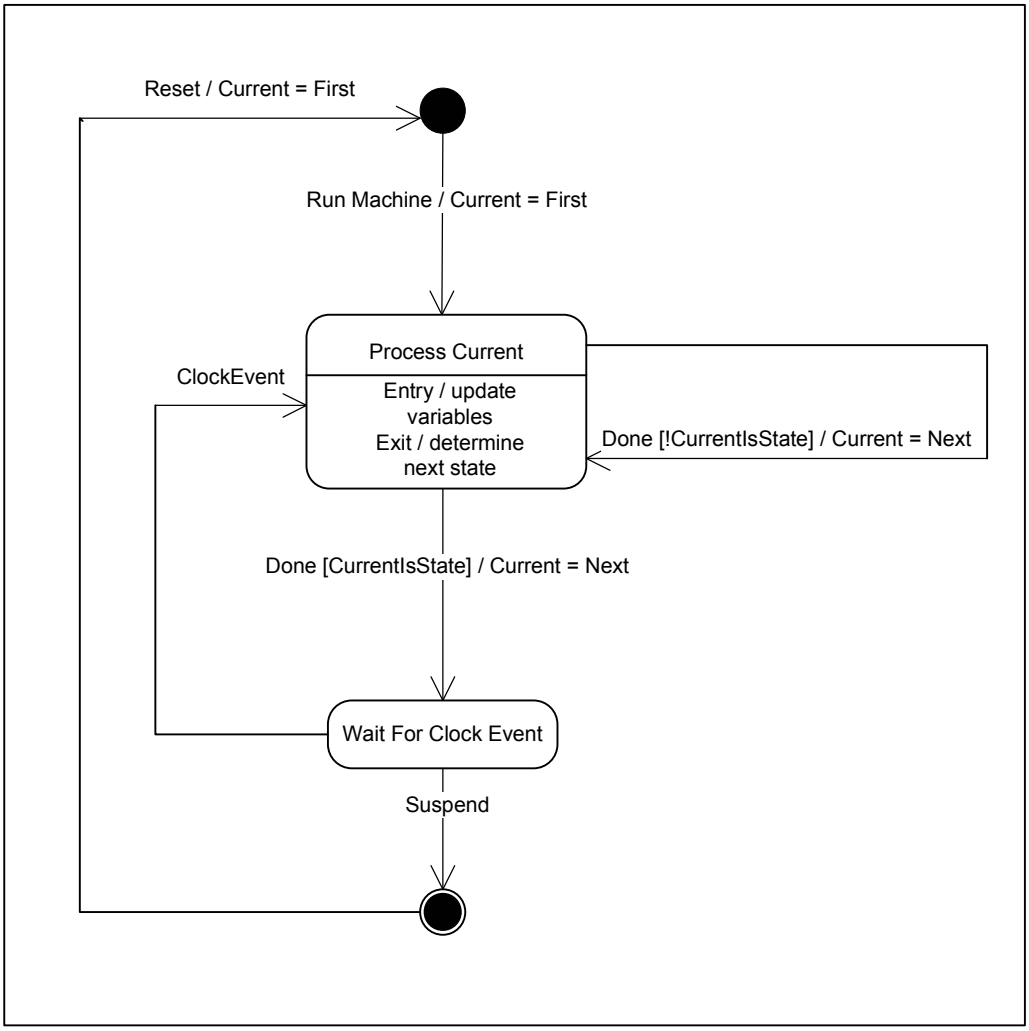


Fig. 6.4.2: ASM Simulator State Chart Diagram

CHAPTER 7

SOFTWARE ACCEPTANCE TESTING

The project results are presented in this chapter in the form of software acceptance testing. The test cases presented here examine each use-case specification to determine whether the developed system meets the functional requirements originally specified. After the test cases were developed, the software was tested by 3rd year electrical and computer engineers at UCT. The results of these tests are also summarised in this chapter. The 3rd year students that tested the system are referred to as "users" in this section.

7.1 Test Case 1: Place Element

7.1.1 Test Case Number

The "place element" test case was assigned test case number TC1.

7.1.2 Description

The user should be able to place an element on the ASM diagram. This test case corresponds to UC1.

7.1.3 Programmer's evaluation

The implementation was successful

7.1.4 Users' comments

The users were satisfied with the implementation. It was suggested that the element toolbar positioned above the ASM diagram be moved to the side of the ASM diagram for greater convenience in placing elements, and to separate tools for placing an element from running the simulation or accessing the file system. Users rejected the idea of changing the method of placing elements to a drag and drop system. The current system allows multiple elements to be placed at once, one after the other without returning to the element toolbar. A drag and drop system would waste time unnecessarily.

7.1.5 Result

The element toolbar was eventually left in its position at the top of the screen to prioritise changes that add functionality to the system.

7.2 Test Case 2: Edit Element Parameters

7.2.1 Test Case Number

The “edit element parameters” test case was assigned test case number TC2.

7.2.2 Description

The user should be able to edit the name of a state and associate variables with an element. This test case corresponds to UC2.

7.2.3 Programmer’s evaluation

The implementation was mostly successful. However, register operations were not implemented due to time constraints. In the implementation, changing the length of the input associated with a decision box results in only the exit paths that were irrelevant to the new input being lost, rather than all exit paths as originally planned in the use-case specification. The exit paths are deleted without prompting the user.

7.2.4 Users’ comments

The users agreed that the implementation was successful.

7.2.5 Result

The changes mentioned in 7.2.3 were made to the use-case.

7.3 Test Case 3: Connect Elements

7.3.1 Test Case Number

The “connect elements” test case was assigned test case number TC3.

7.3.2 Description

The user should be able to connect two elements together. This test case corresponds to UC3.

7.3.3 Programmer’s evaluation

The implementation was successful.

7.3.4 Users’ comments

The users agreed that the implementation was successful.

7.3.5 Result

No changes were made to the original use-case specification.

7.4 Test Case 4: Select Element

7.4.1 Test Case Number

The “edit element parameters” test case was assigned test case number TC4.

7.4.2 Description

The user should be able to move and delete an element.

7.4.3 Programmer’s evaluation

The implementation was successful, with an amended method of operation (see 7.4.5).

7.4.4 Users’ comments

The users were satisfied with the implementation. A suggestion was made that it would be more intuitive to allow users to drag and drop elements to move them rather than left clicking twice.

7.4.5 Result

The drag and drop suggestion was discarded, mostly due to time constraints. During implementation the select element was amended to operate as follows:

The user left clicks on an element to enable movement mode. The user then moves the element by left clicking on an empty space on the diagram. If the user right clicks the movement use-case is cancelled.

To delete an element the user left clicks on the delete toolbar item and an element then left clicks on an element to delete. To cancel delete the user presses right click

7.5 Test Case 5: Add Variable

7.5.1 Test Case Number

The “add variable” test case was assigned test case number TC5.

7.5.2 Description

The user should be able to add a variable to the active variable list, as in UC5.

7.5.3 Programmer’s evaluation

The implementation was successful.

7.5.4 Users’ comments

The users were satisfied with the implementation.

7.5.5 Result

No changes were made to the use-case specification.

7.6 Test Case 6: Edit Variable

7.6.1 Test Case Number

The “edit variable” test case was assigned test case number TC6.

7.6.2 Description

The user should be able to edit a variable in the active variable list, as in UC6.

7.6.3 Programmer’s evaluation

When renaming variables, complications in updating element variable association lists arose. The ability to change variable names was deemed to be a desirable but unnecessary feature and so was discarded. Otherwise the implementation of this use-case was a success.

7.6.4 Users’ comments

The users were satisfied with the implementation.

7.6.5 Result

The changes put forth in 7.6.3 were made to the use-case specification.

7.7 Test Case 7: Delete Variable

7.7.1 Test Case Number

The “delete variable” test case was assigned test case number TC7.

7.7.2 Description

The user should be able to delete a variable in the active variable list, as in UC7.

7.7.3 Programmer’s evaluation

This feature was regarded as being unnecessary and so was discarded. The user is effectively unrestricted in the number of inputs and outputs available to add to the active variables list. The user may add up to 100 variables, more than enough for the purposes of this simulation package.

7.7.4 Users’ comments

The users agreed that the ability to delete variables was unimportant.

7.7.5 Result

This use-case was discarded.

7.8 Test Case 8: Run Machine

7.8.1 Test Case Number

The “run machine” test case was assigned test case number TC8.

7.8.2 Description

The user should be able to run the ASM machine, clocked internally by the system, as in UC8.

7.8.3 Programmer’s evaluation

This feature was regarded as being unnecessary and so was discarded. Given the purpose of the simulation package, it would seem more important for the user to be able to clock the machine manually and thus be allowed the time to view outputs and determine what values to feed into the input manually.

7.8.4 Users’ comments

The users expressed that the run machine facility could be useful and should be implemented in the future.

7.8.5 Result

The run machine use-case was listed for future development.

7.9 Test Case 9: Step Machine

7.9.1 Test Case Number

The “step machine” test case was assigned test case number TC9.

7.9.2 Description

The user should be able to step the ASM forward one clock cycle, as in UC9.

7.9.3 Programmer’s evaluation

The use-case was successful. It was decided during implementation to allow the user to make changes to the ASM diagram (moving elements around, adding elements, altering connections, adding variables etc) while the simulation is in the “suspended” state. The correctness of the ASM diagram (whether all the elements are connected together properly and whether the initial state is defined) is checked on initialisation of each step use-case. Elements that disobey the ASM rules are highlighted and an error messages shows.

7.9.4 Users' comments

The users suggested that if an attempt is made to step the machine without any initial state specified, the "set initial state" dialog should appear for convenience and save time. The users expressed that highlighting elements that disobey ASM rules when the machine is stepped is a useful addition to the use-case.

7.9.5 Result

The users "set initial" suggestion was incorporated into the use-case.

7.10 Test Case 10: Reset Machine

7.10.1 Test Case Number

The "reset machine" test case was assigned test case number TC10.

7.10.2 Description

The user should be able to reset the ASM, as in UC10.

7.10.3 Programmer's evaluation

The implementation was a success.

7.10.4 Users' comments

The users were satisfied that the implementation works.

7.10.5 Result

The implementation of the use-case was as in the specification.

7.11 Test Case 11: Create New ASM Diagram

7.11.1 Test Case Number

The "create new diagram" test case was assigned test case number TC11.

7.11.2 Description

The user should be able to clear the current ASM diagram and create a new one, as in UC11.

7.11.3 Programmer's evaluation

The implementation was a success.

7.11.4 Users' comments

The users were satisfied that the implementation works.

7.11.5 Result

The implementation of the use-case was as in the specification.

7.12 Test Case 12: File System

7.12.1 Test Case Number

The “file system” test case was assigned number TC12.

7.12.2 Description

The user should be able to save the current ASM diagram to permanent storage, and retrieve saved diagrams from storage as in UC12 and UC13.

7.12.3 Programmer’s evaluation

The implementation was incomplete due to unsolved bugs with the file retrieval system.

7.12.4 Users’ comments

The users commented that being able to save and load work is an extremely desirable feature, though not essential.

7.12.5 Result

The implementation of the use-case was listed for future development.

7.13 Test Case 14: Get Help

7.13.1 Test Case Number

The “get help” test case was assigned test case number TC13.

7.13.2 Description

The user should be able to view a readme file for the ASM, as in UC14.

7.13.3 Programmer’s evaluation

The implementation was a success.

7.13.4 Users’ comments

The users were satisfied that the implementation works.

7.13.5 Result

The implementation of the use-case was as in the specification.

7.14 Test Case 14: Set ASM Properties

7.14.1 Test Case Number

The “set ASM properties” test case was assigned number TC14.

7.14.2 Description

The user should be able to set the ASM properties, as in UC15.

7.14.3 Programmer's evaluation

Since the Run ASM use-case was discarded the clock speed feature became obsolete. Thus the "set ASM properties" feature is used exclusively to set the ASM initial state.

7.14.4 Users' comments

The users were satisfied that the implementation works.

7.14.5 Result

Only the "set initial state" part of the use-case was implemented.

7.15 Test Case 15: Other Issues

7.15.1 Efficiency of Execution

The program executed efficiently on white lab and DC lab computers to the satisfaction of both the programmer and users.

7.15.2 Ease of Use of Interface

The interface is sufficiently easy to use, however suggestions were made (outlined in the previous test cases) to make the interface more intuitive and convenient.

CHAPTER 8

CONCLUSIONS AND RECOMMENDATIONS

A graphical user interface (GUI) allowing an ASM diagram to be built and experimented with was successfully implemented. The system effectively verifies the correctness of the ASM constructed on the diagram, in this way the rules of ASM construction are successfully demonstrated. The user is also able to associate elements of the ASM with inputs and outputs. The ASM diagram flow of control can then be manipulated using the inputs. Changes in the outputs can be observed to demonstrate the flow of control of the ASM. In this way the simulation package achieves its primary goal of demonstrating ASM's to afford students a clear understanding of how they work.

The feature set of the final simulation package is rather limited, future development of the software is certainly desirable to make it a more complete package. Listed future developments at this stage are as follows:

- Allow the ASM diagram canvas to be resized and dynamically scrolled to allow for larger ASM's to be built.
- Allow window controls to be resized and moved with respect to each other to customize the appearance of the interface to the user's preference.
- Add state names to the diagram to correspond with standard ASM convention.
- Add a list of elements and variables associated with each to the main window for clarity.
- Implement register operations for states and conditional outputs for a more complete simulation of ASM construction.
- Implement Boolean expressions of inputs for decision boxes.
- Allow the deleting of inputs and outputs.
- Provide a file system to allow diagrams to be saved and restored from disk.
- Develop an extension package to the simulator system that allows a schematic of the digital logic for the system being modelled to be generated directly from the ASM diagram.

References

[1] Mano, M., M., & Kime, C., R., *Logic and computer design fundamentals.* 3rd ed. Upper Saddle River, N.J.:Pearson Education, 2004.

[2] Ambler, S., W., *The object primer: Agile model driven development with UML 2.* 3rd ed. Cambridge University Press, 2004.

[3] Comparison of Java and C++. *Wikipedia, the Free Encyclopedia*. [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B [19 October 2006]

[4] PIGUI. *Wikipedia, the Free Encyclopedia*. [Online]. Available: http://en.wikipedia.org/wiki/PIGUI [19 October 2006]

[5] Object oriented languages: a comparison. *Eifel Software webpage*. [Online]. Available: http://archive.eiffel.com/doc/manuals/technology/oo_comparison/page.html [19 October 2006]

[6] Braem, F., © 2001-2002. *wxWindows: Programming cross-platform GUI applications in C++* [Online]. Avaliable: http://www.wxwindows.org/docs [2 October 2006]

[7] Guthrie, W., 1995. An overview of portable GUI software. *SIGCHI bulletin*. 27(1):55-69.

[8] Gain, J., & Kelleher, J., 2004. Object-Oriented Analysis. *CSC202S lecture notes*. University of Cape Town Computer Science Department.

[9] Rumbaugh J., Jacobson I., & Booch G., *Unified Modeling Language Reference Manual.* Addison Wesley, 1997.

[10] Vernon V., © 2004. *Understanding UML Class Relationships*. Jubatus Corp. [Online] Available: http://www.jubatus.com/publications/articles/ [10 October 2006]