COM1030 X-machines for system specification.

1. Overview.

Recall the purpose of different stages of a project.

1.1 **Requirements**. These describe, in some detail, **what** the system is supposed to do and **how well** it is supposed to do it.
A requirements document is mainly text and should be **understandable** by the client – who has to sign it off.

1.2 A **specification** is a more technical document. It is a translation of the requirements document – or key parts of it – into a technical language that can be used by designers, programmers and testers and maintenance teams. It says what the system has to do, not how it should be designed. A specification could be implemented in many different ways and using many different languages – Java, C, Haskell etc.

The specification can be studied in detail in order to understand any issues that may be problematic when it comes to the coding.
In some case the specification is written using mathematics – it is called a *formal* specification then – and it is a mathematical model of the proposed system. Writing such a document can help you to understand the requirements better. It may also be possible to analyse this model to see if it has any *undesirable* properties such as **unsafe** states. This is important in safety critical applications such as flight control software, nuclear power controllers etc. advanced software can sometimes be used to find these problems – these are called **model checkers** – they can only work if the specification is written in mathematics.

Formal specifications can also be used to gen**erate powerful tests** – this is what we will use here.

It may be possible, one day, to generate code directly from such a specification – this is called cod**e generation**. Some HTML generators can be found in some desktop publishing software.

1.3 **Design** documents. These are more focussed on the implementation and are a guide to programmers on how to create a software architecture to accomplish the implementation. There could be many different ways of making a design from a specification. These could include UML diagrams, for example.

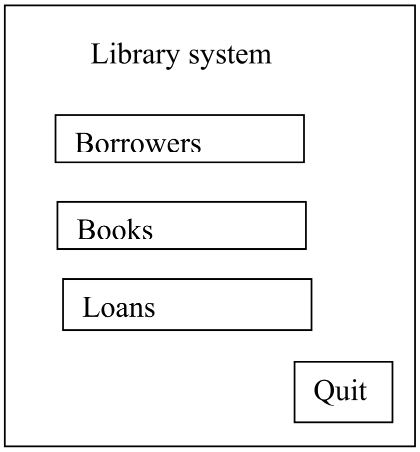
2. Formal specifications.

These have been proposed for the past 20 years and a number of notations and languages have been proposed – Z, VDM, B, etc. None have caught on – they are too inflexible, complicated and hard to use. They do not seem to scale to the required level of complexity for modern day systems.

We have developed a new approach that has been more successful – it is called **X-machines**. It combines the formal descriptions of types that we have been looking at with finite **state machines**.

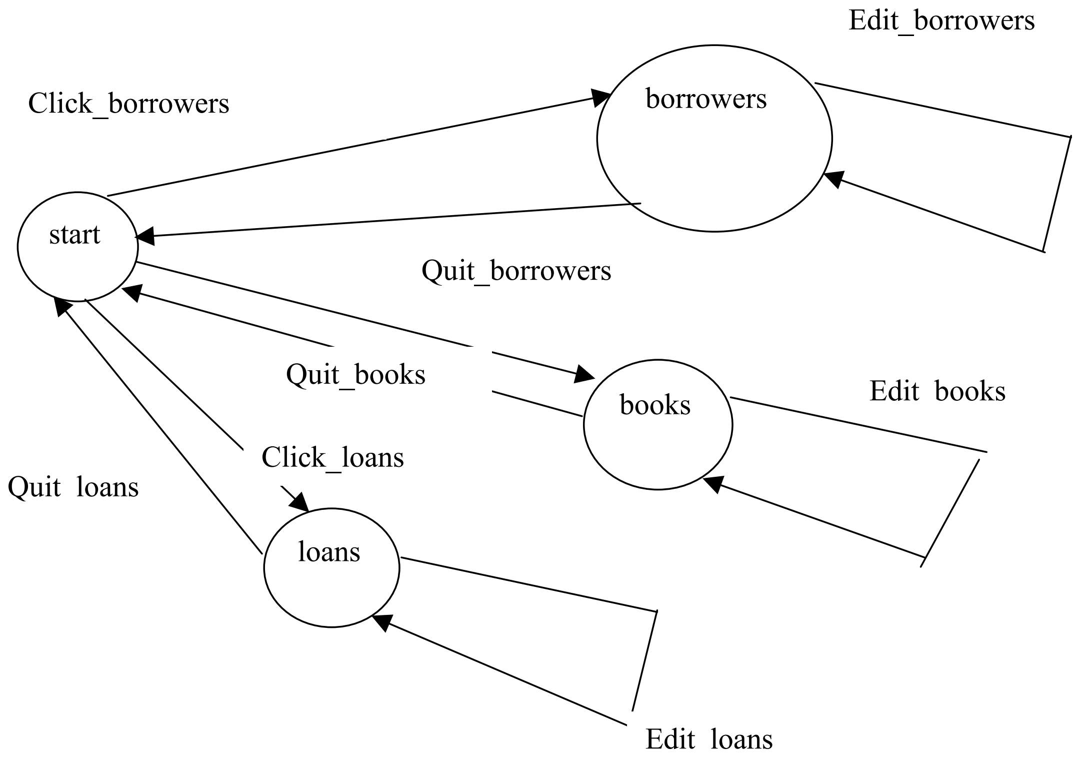
2.1 *System specification of a library system*.

We start with the initial screen display:



Here, the opening screen offers 3 buttons to access the three main subsystems and a Quit button.

The dynamics of the system are described by *a top level systems X-machine*:

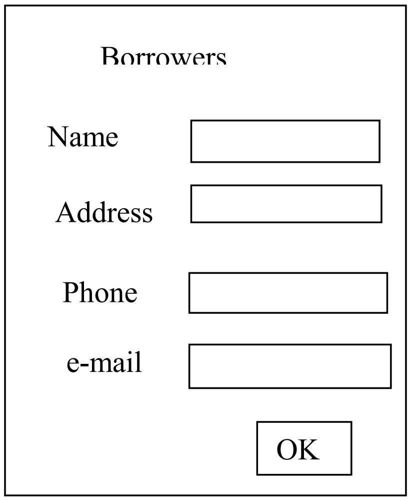


In such a machine we have states: **Start, Borrowers, Books, Loans**;
And transitions: *click_borrowers; click_loans; quit_loans* etc.

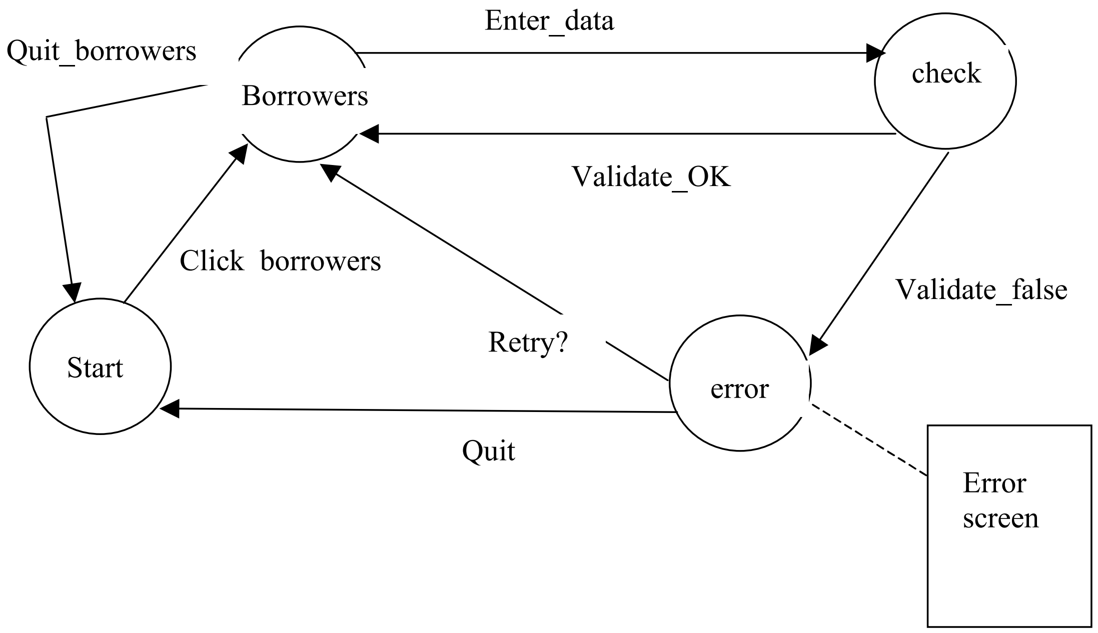
Each transition must have a **start** state (its source) and an **end** state (its target).

This diagram is an X-machine – it consists of a set of **states** and a set of **transitions** that are labelled with **functions** – these functions do things to the system and its data.

Some of these transitions are examples of **events** that users carry out – such as clicking on a button and the result of the transition may be that a new screen appears. Here is a screen for the Borrowers subsystem:



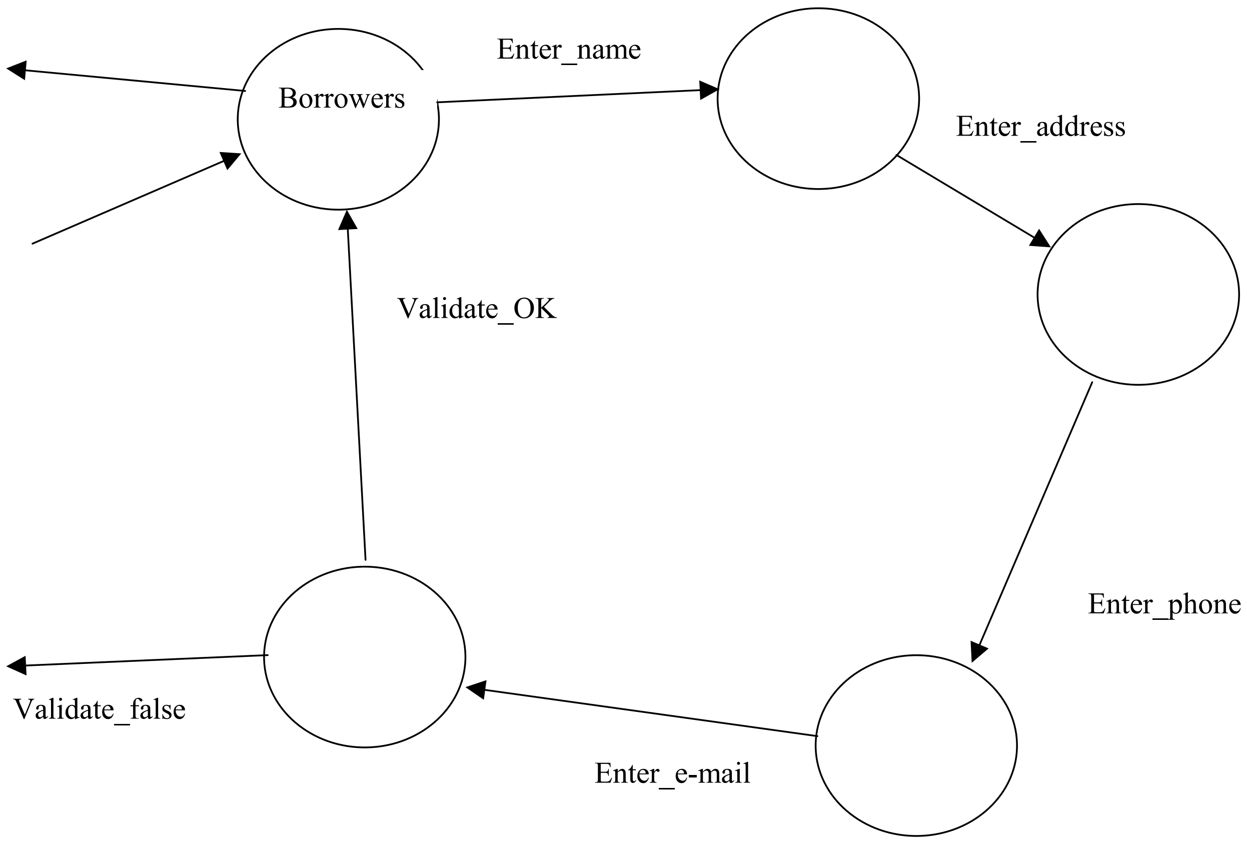
We can expand the *edit_borrowers* function into a lower level X-machine:



If the borrower's details are correctly entered and are not already in the database then the val*idate_OK* transition is taken and the data stored. If there is an error, eg. the borrower is already present in the database, then the *validate_false* transition is taken and this leads to the **error** state.

We must now decide what to do. We could present an error message and offer options to *retry* or to *quit*. The *validate_OK* and *validate_false* transitions are triggered by clicking on the OK button.

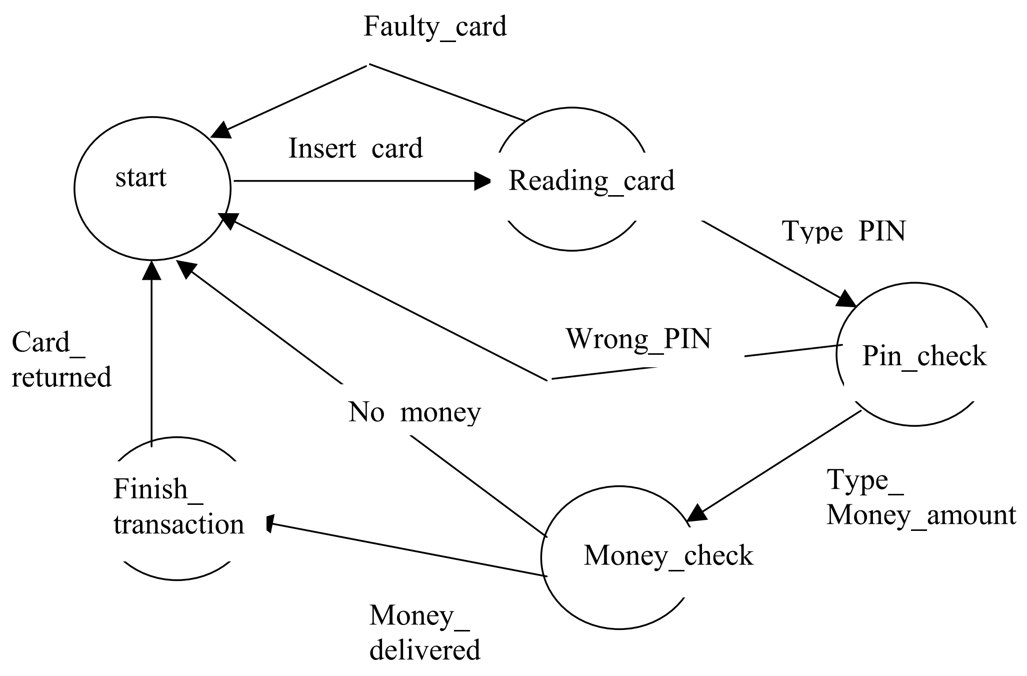
Further refinements are possible:
enter_data is expanded into a lower level submachine:



Note that the state **Borrowers** has 3 arrows in and 1 out, the submachines must match up correctly with the parent machine.

In this model we put the name in first, then the address etc. We can change the order by changing the machine, we could even design the machine to allow for any of the data values to be put in in any order.

2.2 Case study.
Early ATMs (Bank machines) were like this:



The problem was: people were forgetting to take their cards. Why?
The objective of using the system for most people was to get some money. When the money was delivered they had achieved their objective and many then left, leaving their cards behind. The solution was to redesign the interface to give out the card *before* the money. This is done easily by changing some states and transitions around.

2.3 Defining the functions formally.
We need to specify a number of types and the database for the borrowers' details. We have *name, address, phone, e-mail*. All of these are sequences of characters, some just of letters, some just of numbers. A typical record could be:

(John Bull, Regent Court, 07066655544, jb@dcs.shef.ac.uk). For each record the system will automatically generate an Identity number (ID).
Thus the database will contain information like:

0001 → (John Bull, Regent Court, 07066655544, jb@dcs.shef.ac.uk)

The type of the data base is thus:

$$\mathcal{D}_{\text{borrowers}} : \aleph \to (\text{seq}[\text{CHAR}]) \times (\text{seq}[\text{CHAR}]) \times (\text{seq}\ \aleph\ ) \times \text{seq}[\text{CHAR}])$$

Once all of the individual's data has been entered the system has to check whether it is already in the database, so *enter_e-mail* sends a message to the database to check this.

enter_e-mail : ((*name, address, phone, e-mail*), $\mathcal{D}_{\text{borrowers}}$) =

if there is n ● ℵ such that $\mathcal{D}_{\text{borrowers}}$ (n) = (*name, address, phone, e-mail*)

then (‘already present’, $\mathcal{D}_{\text{borrowers}}$)

else (‘OK’, $\mathcal{D}_{\text{borrowers}} \oplus$ (*name, address, phone, e-mail*))

In the first case a screen appears telling us of the failure - ‘already present’ is displayed, it may then ask us whether we wish to *retry* or *quit*.
In the second case the screen is the **borrower** screen and the database is updated with the new information using the next available number for the ID (overriding update operation).

Notation.

Each function is of the form:

Function : (*input, memory*) = (*output, memory’*)

Where the *output* could be a screen display or some other information and, in cases of control systems, an action on some device etc. and *memory’* is the new value of *memory* (note that this may be the same as the previous value).

In some cases it doesn’t matter what the memory is,
Eg. **Click_borrowers**: (*borrowers_click*, --) = (‘*Borrowers’screen*’, --)

In other cases there may be different results depending on the situation;

Function (*input_value, memory_value*) =

if *memory_value* = m_1 then ($output_1$, m_1')
if *memory_value* = m_2 then ($output_2$, m_2')
….
….
etc.

3. An Eclipse tool for X-machines.

A plug-in for Eclipse is available here:

http://www.dcs.shef.ac.uk/~cthomson/xxm/xxmplugin.html

And an animated tutorial is available here:

http://www.dcs.shef.ac.uk/~cthomson/xxm/XXMEclipseTool.htm

Try to use these for stage 2.

WMLH – 28/11/5