

Significance of Models and Modelling Languages

Specify, describe, design and rigorously analyse systems using a *model* of concurrent system.

Good engineering practice

Models created in a formalism (formal system), a theory or a logic.

Models developed using appropriate *modelling* language.

We require a modelling language which captures the inherent features found in concurrent systems.

UML is a well-known modelling language.

Why does concurrent computation differ from sequential computation?

Model a sequential program as a mathematical function over the state of the memory; that is, a function from *old* state to *new* state as the state changes via activities on it through time.

If you know the function for a given program *and* the start state then you can deduce the final state.

But this assumes the program controls the memory and has *sole* control over it.

What if other programs may run concurrently and may also change the contents stored in memory?

The functional model is not adequate. Two programs with the same functional behaviour may behave differently when subjected to the same interference by a third party.

Example (from Robin Milner) Consider 2 programs as follows:

:	:
:	:
x:=1;	x:=0;
:	x:=x+1;
:	:
:	:

These two programs contain program fragments which have the same effect (on memory)

- (a) $x := 1;$
- (b) $x := 0;$
 $x := x + 1;$

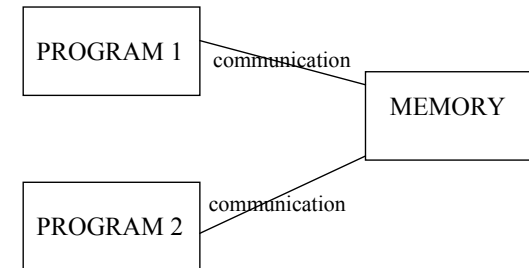
Now let a third program run *concurrently* with that containing (a) and at some unpredictable point in *time* (note the inclusion of time) it will perform $x := 1$ and this assignment will also have an effect on memory.

Irrespective of the exact relative occurrence of the assignments; if x refers to the same location in memory, x will take value 1.

If the program fragment containing $x := 1$ runs concurrently with fragment (b) then the resulting value of x will be either 1 or 2, depending on the *relative* occurrence of the various assignments, i.e. does $x := 1$ occur before or after $x := 0$?

When we have concurrent computation all parties, including the memory, can interact. Hence programs and memories are all "first class" objects: neither is completely subservient to the other.

What we have is the following simple network:



All 3 components (or agents) are potentially concurrently active.

Why do we wish to rigorously *model* concurrent systems?

Subtle behaviour, as illustrated in above example.

Large number of concurrently active components e.g. process or integrated circuit chip has very complex behaviour although this is achieved by combining many simple parts – transistors and logic gates.

The need to determine *correctness*: for certain critical applications this is *very* significant. e.g. nuclear powerstation control system, avionics systems.

Generally, assurance that a system behaves as intended is important. With certain systems, incorrect design can cause catastrophic failure – *safety critical systems*.

Design techniques for complex, concurrent systems. Today's systems are becoming even more complex. Look at integrated hardware and software systems, embedded systems, real time systems. Why is it difficult to conceptualise concurrent behaviour? Designers generally think sequentially.

Designing a new system, such as complex concurrent software, and describing an existing one, such as computer network, are clearly related tasks.

Modelling for Systems Design

Need to develop practical methodologies for modelling concurrent systems.

Need an appropriate formalism for this.

Modelling as key part of the software design process.

Two approaches:

ASSERTIONAL

logic
statement of facts
temporal or modal logics

EVENT BASED

transition systems
actions and states
dynamic formalisms

We will use the *Circal formalism*, an event/state formalism.
Practical experience of using the *XCircal description language*.

This embeds Circal in language constructs which support the creation of accurate descriptions or models of real world phenomena/systems/interacting agents.

Use of the *Circal System* to build system descriptions and then analyse them (either by simulation or by formal proof).

Aim to introduce fundamental concepts of concurrency using Circal.

Modelling Structure and Behaviour

Basic Ideas:

We wish to create models which capture the behaviour of *systems*. A system is constructed from a number of active objects or *agents* or *processes*. Each has *behaviour*, that is, it performs *actions* or *events* through *time*.

A system also has *structure*. This reflects the connective links between the individual component agents which comprises the whole system. Each agent corresponds to a *node* and each link to an *arc* in a mathematical device called a *graph*.

Systems may be *constructed hierarchically* where agents are *composed* or *linked together* to form a system, and sub-systems composed together to form larger systems. Certain communication links may be *hidden* or *abstracted* and the resulting system then treated as a single agent. We say that the system has been “black boxed”.

We describe system behaviour in a constructive (or algebraic) manner using suitable *behavioural composition* techniques, reflecting the structural hierarchy (or just structure) of the system.

Automata:

We focus on the concept of a *finite automata* as the basis of behavioural description. Each automaton will perform *events* together with (or in partnership with) the other automata which it is connected to. That is, automata *interact* with each other. These events occur in *time* and we use techniques to model time. Hence we can capture *when* “things” happen as well as *where* events occur i.e. on the communication links joining two particular.

Interacting Automata:

In this course we focus on a particular modelling approach based on interacting automata. A *process* is analogous to a finite automata or finite state machine and captures the event-based behaviour of some artifact such as a distinct unit of software. Such a *formalism* may be presented algebraically i.e. as an algebra in terms of *operators* and *operands*. In our case the operands are events which occur as part of an interaction with another automata/process. The operators are used to *construct* the processes from operands. Hence we describe a system in terms of a suitable construction of the processes which describe its component parts.

This is the inherent constructive or algebraic approach. It allows us to manage *system complexity* by constructing system descriptions in a *structured*, constructive manner. That is, build a complex device using previously built *components*.

State:

As with other formalisms based on automata theory, the concept of *state* is fundamental. As events occur a process/automaton will (or may) make a *transition* from the current state to the next state. Hence states *and* events are core concepts.

We will see that this process algebra approach is well-suited to modelling *event occurrence* via interaction and timing (i.e. *when* an event occurs).

Finite Automata:

Finite automata are models of agents that exist in a finite number of *states*. A finite automaton models behaviour by moving from state to state using a predetermined function which defines the transitions between states and the *events* that cause the transitions. A set of events called the *input* is processed in order, until no more inputs are left. A finite automaton can be represented by a *state table* mapping states and events to states or diagrammatically as a *state transition diagram*. Note that a finite automata is similar to a finite state machine.

Definition

A finite automaton A is formally specified by the 5-tuple

$A = \{S, \Sigma, t, s_0, F\}$ where:

1. S is a set of states,
2. Σ is the input alphabet (a set of events),
3. $t: S \times \Sigma \rightarrow S$ is a transition function,
4. s_0 is the initial state, and
5. $F \subseteq S$ is a set of *accepting states*.

State Table

current state	next state		
	input		
	0	1	2
s_0	s_0	s_1	s_2
s_1	s_2	s_1	s_1
s_2	s_3	s_1	s_2
s_3	s_3	s_3	s_3

State Transition Diagram

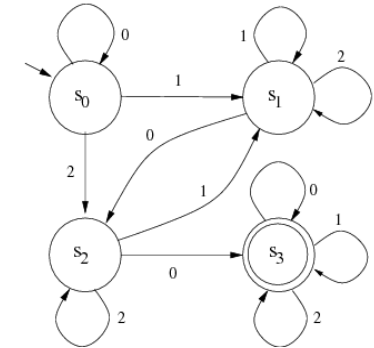


Figure 1

Figure 1 gives an example state table and state transition diagram for the same finite automaton. The finite automaton in the figure has states $S = \{S_0, S_1, S_2, S_3\}$ and input alphabet $\Sigma = \{0,1,2\}$. Circles represent the states and the lines between them are the events allowing transition (i.e. an arrow) between states. The state transition diagram specifies the finite automaton completely. The initial state is represented in the state transition diagram by the unlabelled transition terminating at the initial state. Accepting states are denoted in the state transition diagram by double circles. When a finite automaton's operation ends at an accepting state then the automaton is said to have recognized the input string. In this case, the string 21001 is an example of a string that is accepted by the automaton.

Modelling Terminology

The word model is used to mean a representation of the *behaviour* of some physical object. For physical system such as the flow of a fluid the dynamics of a model may be specified by differential equations. Thus the *language* of differential equations is used to describe the changing behaviour over time of some artifact or object such as the weather, a moving air mass. Thus, a particular mathematical theory is then used to capture a specific temporal behaviour.

We do not use continuous temporal (that is, related to time) models such as differential equations. We use discrete, event-based models. Models constructed using interacting automata theory allow “virtual” worlds to be examined and analysed via *simulation*.

Suitable models of a physical system are constructed within an appropriate formal theory or *formalism*, which usually involves a description language. These models are then simulated using a simulation engine, which *exercises* or *runs* models as if they were programs. The states of the component automata of a system model vary with time, as they interact. Patterns of state values through time are the “outputs” of most interest. The occurrence of events between automata (their interactions) are also significant and may be used to analyse the physical world via its virtual representation.

Interaction and Communication

The concurrent program example used earlier indicated that the *interaction* between concurrently active components is a key concept in concurrent systems.

Interaction and *communication* are intimately bound together.

We are therefore interested in modelling how components of a concurrent system communicate between themselves, and also how the system communicates with the external *environment*.

Process calculi, such as Circal (Milne), CCS (Robin Milner) or CSP (Tony Hoare), aim to be *primitive* formalisms.

What is meant by this is that they contain features which capture (some or all of) the primitive underlying concepts found in a concurrent system. They aim to maintain a rich modelling potential and not restrict the user to a predefined modelling style or approach to modelling. They are *powerful* yet *primitive*.

Communication

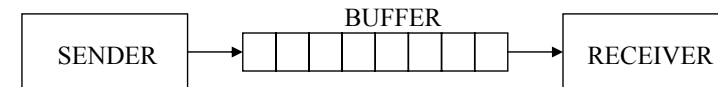
Consider various communication mechanisms.



Assuming that SENDER can always send a message (provided the ether is not full) and RECEIVER can always receive (provided the ether is not empty) then the ether can have various distinct characteristics:

1. infinite, unordered capacity
2. finite, unordered
3. infinite, ordered
4. finite, ordered

The latter corresponds to a bounded buffer and may be pictured as:



We can see that there are many types of communication media, many more than the 4 used above. When creating a modelling formalism we cannot constrain a user to modelling only one, as which one should we choose as the preferred model?

We use a more primitive notion of communication. If we wish to model one of the many types of *communication media* then we should do this explicitly using the underlying primitive communication constructs. The media itself will be yet another concurrently active component in the system which will communicate with the other components, the SENDER and the RECEIVER in this case, using the primitive communication construct.

In the picture above we would then have a system of three interacting agents (modelled by three interacting automata), namely SENDER, RECEIVER and BUFFER.

What do we take as this primitive communication notion? First, let us examine the idea of an action or event.

From a modelling perspective let us assume that an action has the following properties:

1. an action occurs instantaneously;
2. a singleton action cannot be subdivided into anything more primitive;
3. occurrence of an action is a *cooperative* activity between 2 or more components.

Actions are the modelling primitive which we use to *communicate* information between components.

This communication will involve two or more components *synchronizing* to perform the required action.

Structure

We have informally introduced the concept of components communicating over *links* via individual actions.

For example, suppose we take a cell component and *construct* a simple system involving two of these cells, as in:



The arrow indicates that one cell may communicate with another, and that this communication is directional. That is, CELL1 *generates* an action and communicates it to CELL2. This action is actually an *interaction* in that the two components *participate* in the action occurrence. This then creates the idea that with respect to a particular communication, the two (or more) participants have a different role to play:

- one is the *active* sender;
- the other is the *passive* receiver.

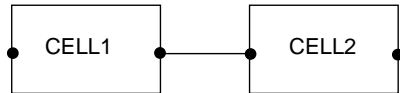
Most concurrent systems are not *closed*; that is, most will interact with some surrounding *environment*.

Suppose our "two cell" system is to be used as a bounded (by 2) buffer. Then it must be able to accept information to be stored via CELL1, and output to the outside world via CELL2.

Ports

It is convenient to picture this as follows:

The left hand port (the black dot) of CELL1 will be used as an input with which to interact

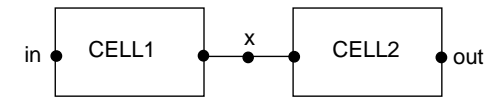


with the environment. It will then pass this information over to CELL2 (since a buffer must have some mechanism for passing information "along") and this will subsequently be output on the righthand port of CELL2.

Notice that the ports above are not distinguished as input or output. It will be the description or model of cell behaviour which determines how the ports are used to perform actions. This will also determine whether they are being used for input or output.

Port Labels

In the above diagram it is difficult to try to identify one port from another. Let us *label* them to give them identities:



Notice that the *communication* link between the two cell components is also labelled, by x.

We now have two types of identifiers, port labels and component names. We will generally use lower case for the port labels and upper case for the component names.

Notice too that we have dropped the use of an arrow. In fact the description of the component cells will determine which ports are used as input or as output and hence the directionality on the communication links. In certain modelling scenarios there may not even be a directionality concept; here all components will participate in a "pure", non-directional interaction. Similarly, some ports may be used in different directions at different points in time; bidirectional pass transistors are a good example.

SYSTEM STRUCTURE:

The structure of the 2-cell system seen above consists of:

- 2 *named* processes;
- a *named* communication link called x joining them together;
- 2 external links, named *in* and *out*.

Such a structure is related to a mathematical construct known as a graph.

The components correspond to *nodes* or *vertices* of the graph, and the linkages correspond to graph *edges*.