Petri nets and STGs:

- Notation
- Modelling techniques
- Decomposition

Victor.Khomenko@ncl.ac.uk
From FSMs to PNs

- Represent a concurrent system using several FSMs, each with its own local state, communicating by jointly executing common actions.

- The overall system state is represented in a distributed manner, as a collection of the local states of the individual FSMs (in contrast to a single currently active state of an FSM).

- The system’s structure and potential changes of states can be visualised using a directed graph.
Example: Producer/Buffer/Consumer

begin parallel

Producer:  
while true do
  produce item
  deposit item

Buffer:  
while true do
  deposit item
  remove item

Consumer:  
while true do
  remove item
  consume item

end parallel
Modelling Producer

**Producer:**

```plaintext
while true do
  produce item
  deposit item
```

**FSM model:**

```
prod  dep
```

**PN model:**

```
prod  dep
```
Modelling Buffer

Buffer:  

while true do
    deposit item
    remove item

FSM model:

PN model:
Modelling Consumer

Consumer: \textbf{while} true \textbf{do}
\begin{itemize}
  \item remove item
  \item consume item
\end{itemize}

FSM model:

\begin{tikzpicture}
  \node[state,initial] (rem) {$rem$};
  \node[state] (cons) {$cons$};
  \draw (rem) edge (cons);
\end{tikzpicture}

PN model:

\begin{tikzpicture}
  \node[state,initial] (rem) {$rem$};
  \node[state] (cons) {$cons$};
  \path (rem) edge (cons);
\end{tikzpicture}
Composing Petri net models

Draw the three Petri nets next to each other and glue together boxes with the same label (synchronous communication = join execution of common actions); some (informal) annotation can optionally be added:
Composing Petri net models

Draw the three Petri nets next to each other and glue together boxes with the same label (synchronous communication = join execution of common actions); some (informal) annotation can optionally be added:
Structure of Petri nets

Interpretation of different components:

- circles are called **places** and represent local states
- boxes are called **transitions** and change (perhaps several) local states; transitions may be **labelled** by some actions
- black dots are called **tokens** and represent the current holding of local states (in general, a place may contain several tokens – e.g. to model a counter)
- **arcs** indicate how executing a transition modifies the state of a Petri net
Transition firing rule

Consider a PN with some marking $M: P \rightarrow \{0, 1, 2, \ldots\}$.

A marking $M$ enables a transition $t$ if all its preceding places are marked. **Firing** an enabled transition $t$ removes a token from each preceding place and then adds a token to each succeeding place.
Firing sequences and reachable markings

Notation: $M[t]M'$ means that transition $t$ is enabled at marking $M$, and firing it leads to marking $M'$

Starting from the initial marking $M_0$ we may fire a sequence of transitions $t_1t_2...t_n$ if there are markings $M_1,M_2,...,M_n$ such that $M_0[t_1]M_1[t_2]M_2...M_{n-1}[t_n]M_n$

Then $fs = \ell(t_1)\ell(t_2)...\ell(t_n)$ is a **firing sequence**, and $M_n$ is a **reachable marking**

Firing sequences represent what an **observer** of the system would see, and reachable markings are the ones the system can find itself in. In the FSM terminology, firing sequences are strings generated/accepted by a PN.
Example
Firing sequence: prod dep rem
Short-hand drawing conventions

- Omit drawing boxes for transitions – draw just labels:
  - If a place has one incoming and one outgoing arc, just draw an arc through it:
    - Colours of the labels: input, output, internal; the latter behave like outputs but are ignored by the environment.
Short-hand drawing conventions: examples

- Producer/Buffer/Consumer system:

  ![Diagram of Producer/Buffer/Consumer system]

- Inverter circuit:

  ![Diagram of Inverter circuit]
Modelling techniques: basic

Sequential execution of actions

Parallel execution of actions

Synchronisation
(joint execution of an action)

Testing a condition without consuming a token
The presence of a token without consuming it is common. Instead of drawing two arcs going in opposite directions one can either draw a single arc with two arrowheads, or just a line without arrowheads, so the following three drawings have the same meaning. Such arcs are called **read arcs**.
Modelling techniques: complementary places

- Places p and q in are **complementary** iff any transition removing a token from one of them puts a token to the other.
- If p can contain at most one token then creating place q complementary to p such that q is initially marked iff p isn’t does not change the behaviour of the PN.
- Can use a complementary place to **test the negation of a condition:**
Modelling techniques: synchronisation

Example: producer / 3-slot buffer / consumer:

Idea: synchronise the rem action of a slot with the dep action of the next slot:

![Diagram of synchronisation](image)
State Space Explosion

Petri nets can be exponentially more compact than FSMs, in particular for highly concurrent systems, e.g. the FSM for the 3-slot buffer is much larger:
Modelling techniques: counters

**Idea:** use multiple tokens on a place to represent a counter

**Example:** producer / buffer of capacity 3 / consumer:

![Diagram of buffer system with tokens](image)

- **prod** (producer) and **cons** (consumer) indicate the number of tokens in the buffer.
- **dep** (depletion) and **rem** (removal) show the number of tokens removed.

**Note:** the ‘identities’ of items in the buffer are lost!
Modelling techniques: OR-causality

Motivation: quickly react to any of several stimuli

Assumption: all stimuli are eventually provided (if not, still can use OR-causality, but arbitration will be required in the ‘reset’ phase)
Modelling techniques: choices

**Free choice**

- choice
- a
- merge
- b

**Asymmetric choice**

- condition
- choice
- a
- merge
- b

**Controlled Choice**

- choice
- controlled
- a
- merge
- b

**Arbitration**

- acquire$_1$ → a → release$_1$
- acquire$_2$ → b → release$_2$
Modelling techniques: choices

Pooh always liked a little something at eleven o’clock in the morning, and he was very glad to see Rabbit getting out the plates and mugs; and when Rabbit said, “Honey or condensed milk with your bread?” he was so excited that he said, “Both,” and then, so as not to seem greedy, he added, “But don’t bother about the bread, please.”

A. A. Milne, “Winnie-the-Pooh”
Modelling techniques: choices

Why are there any choices at all?

- **Abstraction** of the environment (do not want a detailed model of the rest of the Universe!) – the ‘implementation’ of the environment may well be without choices

- **Resource contention**: have to arbitrate between several clients trying to use the same resource

- **Structural choices**: not ‘semantical’, e.g. due to modelling concurrency by interleaving; can be removed by restructuring the specification

- **Non-deterministic choices**: either not ‘real’ and can be removed (by determinising the specification or making OR-causality explicit) or indicate that the system does not have enough information to perform its function (e.g. more inputs from the environment are required)
Modelling techniques: inputs and outputs

- Often actions are partitioned into inputs that are performed by the environment and outputs that the system has to produce (and perhaps internal actions)

- Then the PN can be viewed as a contract between the system and its environment:
  - if an input is enabled than the environment is allowed (but not obliged) to produce it; the environment must not send any inputs that are not enabled
  - if an output is enabled then the system is obliged to produce it eventually (or it is eventually disabled – such scenarios can be tricky though); the system must not produce any outputs that are not enabled
Modelling techniques: inputs and outputs

- **Example:** inverter circuit

![Diagram of inverter circuit]

- Initial state: \( i=0, \ o=1 \)
- Initially \( i^+ \) is enabled, i.e. the environment may (but does not have to) change the value of input to 1
- After \( i^+ \) fires, \( o^- \) becomes enabled, i.e. the inverter must eventually change the value of its output to 0
- Meanwhile, the environment is obliged not to change the value of the input (no input transition is enabled) – it must wait for \( o^- \) before doing that
Modelling techniques: i/o and choices

- **Input / input** choices: usually appear due to abstraction of the environment

  **Example:** Vending machine models the user’s behaviour as a free choice between chocolate and coke. The real user might well have no choice (e.g. wants a drink, not a snack). However, a detailed model of the user would be infeasible, so this free choice overapproximates the relevant (from the vending machine’s point of view) part of user’s behaviour.

- **Example:** Memory circuit handles read and write requests from the CPU. This can be modelled / overapproximated by a free choice, to avoid detailed modelling of the CPU, which might not have a choice what request to send.
Modelling techniques: i/o and choices

- **Output / output** choices: usually due to *arbitration* between clients contending for a shared resource, e.g.:
  - which of two threads gets hold of the mutex first?
  - which of the two requests for a device gets granted first?
  - should a signal from the environment be processed in the current clock cycle or in the next one?

- If clients’ requests arrive too close in time, the system has to make an *arbitrary decision* (cf. *Buridan's ass*)

- In circuits, this leads to *metastability*, which cannot be resolved in bounded time
Metastability

Stable: logic 0

Stable: logic 1

Metastable
Metastability

- Occasionally the system has to make an **arbitrary** decision, i.e. **either** alternative is acceptable, but a choice has to be made

- Metastability can persist for an indefinitely long time!
  - there is a non-zero probability that Buridan's ass will starve to death

- Issues:
  - though the probability of a long delay is small, when repeated sufficiently many times, a nasty scenario **will** happen, and **will** cause malfunction in some kinds of systems, in particular synchronous (clocked) circuits – e.g. when this delay gets longer than a clock cycle (MTBF can be calculated for such systems, and there are ways to trade off performance for MTBF)
  - need to contain metastability (which is analogue by nature) – it must not propagate to the digital part of the system!
Metastability in circuits

- Synchronous (clocked) circuits: need to arbitrate between a clock edge and an input from the environment; a synchroniser is used (may fail occasionally – MTBF):

- Asynchronous circuits: Mutex Element [Seitz, 1979] is used (never fails but may take indefinitely long to resolve, so try to remove from the critical path):
Modelling techniques: i/o and choices

- **Input / output** choices: very problematic, usually indicate a mistake in the design or lack of relevant information / behaviour

- Intuitively, the system and the environment have to make a consistent decision in a **distributed** manner; this cannot be implemented without allowing for the possibility that both actions are performed!

- If such a decision is really necessary and the system collaborates with the environment then it can be delegated to one of the parties (or to a 3\textsuperscript{rd} party), which will make a **local** decision (**output/output** choice) and notify the other party (which will see it as an **input/input** choice)
Exception (?): WAIT element

- WAIT element has a read-consume input / output choice (or rather input / internal choice – but the difference is not important here)

- Upon activation by ctrl+, waits for sig=1 (may ignore short spikes) and latches it as ‘clean’ san

- ‘Clean’ ctrl / san handshake controlled by ‘dirty’ sig
Exception (?): WAIT element

- Determinised and minimised state graph with an optional output
Exception: WAIT element (cont’d)

- Implementation

- The bubble on sig input can be detached as an inverter

Modelling techniques: structural choices

Some structural choices are not semantical, e.g. due to concurrency being modelled by interleaving; such choices can (and should!) be eliminated:
Modelling techniques: non-deterministic choices (benign case)

The lower branch is ‘subsumed’ by the upper one and so can be removed:
Modelling techniques: non-deterministic choices (malignant case)

- If too much is hidden, cannot determinise / implement:

- In this case can hide either a or b, but not both, as these signals decide whether x or y is eventually output
Modelling techniques: parallel composition

- Large systems are not monolithic: they are designed by composing smaller blocks, which in turn are composed from even smaller blocks, etc.

- **Compositionality**: Semantics / behaviour of a composed system must be easily predictable from the semantics / behaviour of its constituent blocks.

- **Examples**:
  - Producer/Buffer/Consumer system
  - Constructing an algebraic expression by e.g. adding two simpler expressions
  - Constructing a digital circuit by connecting smaller circuits with wires
Modelling techniques: parallel composition

- **Problem statement:** Given several PNs that share actions, construct their **parallel composition** – a PN that models the overall behaviour, provided that the shared actions are executed jointly:

\[
\text{PN}_1 \mid \text{PN}_2 \mid \ldots \mid \text{PN}_k
\]

- **Desired properties:** The order of composition should not matter, i.e. ‘|’ is commutative and associative:
  - \( \text{PN}_1 \mid \text{PN}_2 = \text{PN}_2 \mid \text{PN}_1 \)
  - \( (\text{PN}_1 \mid \text{PN}_2) \mid \text{PN}_3 = \text{PN}_1 \mid (\text{PN}_2 \mid \text{PN}_3) \)

- Hence composing several PNs reduces to a number of binary compositions
Modelling techniques: parallel composition

- **Idea:** Glue transitions from different PNs that have the same label (if PNs have several transitions labelled with e.g. $a$, glue each such transition in PN$_1$ with each such transition in PN$_2$)
Modelling techniques: parallel composition

Example:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>

```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>

```

```
= 

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>

```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>
```

```
```

```
= 

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Modelling techniques: parallel composition

When actions are divided into outputs / inputs / internal:

- An action can be an output of at most one PN!
- An action can be an input of 0 or more PNs; gluing two input transitions produces an input transition
- An output of a PN can be an input of 0 or more other PNs; gluing an input transition with an output transition produces an output transition
- The outputs of the composition are all the outputs of all the components
- Internal actions cannot be shared (can rename them if necessary to avoid name clashes), and so their transitions are never glued
Modelling techniques: parallel composition

- Example:
Decomposition

- Logic decomposition: splitting large complex-gates to be able to map them

- System decomposition (our focus): architectural-level decomposition of a system, so that (small) blocks rather than the whole (monolithic) system have to be specified as STGs and synthesised into circuits
  - creative process – but some people are good at it
  - often many sensible ways to decompose
  - blocks with more than ~10 signals are often difficult to design, so need to keep decomposing

- Decomposition is the inverse of parallel composition
  - A bit like factorisation of integers is the inverse of multiplication
Example

Design a circuit that waits for $a$ and then produces $b$ followed by $c$.

Let's pretend we don't know the answer yet.
Communication

- Handshakes – usually 4-phase; can load both phases
- Dual-rail req & single-rail ack to pass some info to the called block (e.g. read/write mode, etc.)
- Single-rail req & dual-rail ack to return some information to the caller
- Token-ring:

```
inv  stage1  stage2  ...  stageN
```

- Exotic stuff like m-of-n for long links
Tricks

- Sometimes the block is still large-ish and difficult to decompose; so need some tricks to reduce the number of signals and the size of the STG.

- Handshake compression: replace \texttt{req+} → \texttt{ack+} and \texttt{req-} → \texttt{ack-} by \texttt{sig+} and \texttt{sig-}.
  - e.g. to insert a WAIT element: use \texttt{w+} instead of \texttt{ctrl+} → \texttt{san+} and \texttt{w-} instead of \texttt{ctrl-} → \texttt{san-}.

- Don’t insert \texttt{sig-} unless it’s ‘loaded’.
  - will violate consistency
  - Petrify can add \texttt{sig-} automatically (unsupported by GUI yet)
  - e.g. for WAIT element, don’t insert \texttt{w-}.
Re-synthesis

- Decomposition creates overheads due to new signals for communication
- If these overheads are not acceptable (e.g. handshake on a critical path) they can sometimes be reduced by re-synthesis:
  - compute the parallel compositing of two or more blocks, usually tightly coupled ones (supported by Workcraft GUI)
  - hide the handshake signals by turning them to dummies (can be combined with composition)
  - optionally contract the resulting dummy transitions
  - synthesise the resulting STG
Beware of computational interference!

- Parallel composition at the level of STGs almost corresponds to connecting circuit blocks by wires…
- …as long as STG contracts are respected
- **Computational interference** happens when one block produces a signal that the other block doesn’t expect; this is masked by STG decomposition
- **Conformation** is the formal verification property stating that computational interference cannot happen
Beware of computational interference!

Example:

Note that the 1\textsuperscript{st} block can produce $c+$ breaking the 2\textsuperscript{nd} block that is still waiting for $b+$, and this can happen at the circuit level.

The composed STG masks this behaviour, i.e. does not correspond to the behaviour of the circuit!
Example: Multiphase buck control
Example: Multiphase buck control (cont’d)
Example: Multiphase buck control (cont’d)
Example: Multiphase buck control (cont’d)

- synthesised SI components
- A2A interface components
- external delay elements
Example: Multiphase buck control (cont’d)