Engineering distributed software?

◆ **Structure**
  Programming-in-the-small Vs Programming-in-the-large
  deRemer and Kron, TSE 1975

◆ **Composition**
  “Having divided to conquer, we must reunite to rule”
  Jackson, CompEuro 1990
How? our approach is .... Model Based Design

integrate modelling into the software lifecycle:
Software Architectures of components, translatable to models

Relatively easy to learn and use:
State Machines in form of LTS (Labelled Transition Systems)

Lightweight Tool support:
Model Checking in form of CRA (Compositional Reachability Analysis) with animation
Concurrency: State Models & Java Programs

Jeff Magee & Jeff Kramer

WILEY

2006 (2nd edition)
http://www.doc.ic.ac.uk/~jnm/book/

- Java examples and demonstration programs
- State models for the examples
- Labelled Transition System Analyser (LTSA) for modelling concurrency, model animation and model property checking.
Chapter 1. Context and experience

Software Architectures
Component types have one or more interfaces. An interface is simply a set of names referring to actions in a specification or services in an implementation, provided or required by the component.

Systems / composite components are composed hierarchically by component instantiation and interface binding.
construction view - **Koala** ADL for consumer electronics

... based on **Darwin**, in use by Philips for product families (IEEE Computer 2000)
“It turns out to be very simple to make different configurations. We are profiting from the **composability** in that it is very easy to create small environments in which to test parts of the software.”

The individual processes are really quite simple state machines. What we really need is a way to compose these state machines, and perform some sort of analysis on the composition…”

Rob van Ommering
Philips Research Eindhoven
Architectural description - multiple views

Structural View

Construction View

Behavioural View

Component implementations

Component behaviour models

Implementation

Modelling and Analysis
Chapter 2. Modelling processes

Primitive components
**Concepts:** component processes  
- units of sequential execution.

**Models:** finite state processes (FSP)  
to model processes as sequences of actions.  
labelled transition systems (LTS)  
to analyse, display and animate behavior.

**Practice:** Java threads
Component/Process:

DRINKS = ( red->coffee->DRINKS \\
| blue->tea->DRINKS \\
).

FSP to model behaviour of the drinks machine:

LTS:
FSP - guarded actions

component COUNT {
    provides inc; dec;
}

COUNT (N=3) = COUNT[0],
COUNT[i:0..N] = ( when (i<N) inc->COUNT[i+1] | when (i>0) dec->COUNT[i-1] )

when guarded choice
process parameter
Process labelling
A countdown timer which beeps after N ticks, or can be stopped.

Java Demo

```java
component COUNTDOWN {
  provides start; stop;
  requires beep;
}
```
A countdown timer

COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
    ( when(i>0) tick->COUNTDOWN[i-1]
    | when(i==0) beep->STOP
    | stop->STOP
    ).
component PERSON - behaviour

component PERSON {
    requires enter; exit;
}

PERSON = ( enter -> bathe -> exit -> PERSON ) @{enter,exit}.

@interface

Actions {enter,exit} are exposed, bathe is hidden.
Labelled transition system LTS:

LTS Animation can be used to step through the actions to test specific scenarios.

PERSON can be minimised with respect to Milner’s observational equivalence.
component BATH - behaviour

const Max = 3
range Int = 0..Max

BATH(N=Max) = BATH[N],
BATH[v:Int] = ( when(v>0) enter-> BATH[v-1] |
              when(v<N) exit -> BATH[v+1] )

Primitive Components - summary

- Component behaviour is modelled using Labelled Transition Systems (LTS).
- Primitive components are described as finite state processes (FSP) using the dynamic operators of the process algebra:
  - action prefix ->
  - (guarded) choice |
  - recursion
- Interface @ represents an action (or set of actions) in which the component can engage (ie. constrains the visible alphabet of the process).
Chapter 3. Modelling systems

Composite components
Concurrent execution

**Concepts**: processes - concurrent execution and interleaving. 
process interaction.

**Models**: parallel composition of asynchronous processes  
- interleaving interaction - shared actions

**Practice**: Multithreaded Java programs
Definition

✿ **Concurrency**
- *Logically* simultaneous processing.

Does not imply multiple processing elements. Requires interleaved execution on a single processing element.
Modeling Concurrency

◆ How should we model process execution speed?
  ● arbitrary speed
    (we abstract away time)

◆ How do we model concurrency?
  ● arbitrary relative order of actions from different processes
    (interleaving but preservation of each process order)

◆ What is the result?
  ● provides a general model independent of scheduling
    (asynchronous model of execution)
If P and Q are processes then \((P||Q)\) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

\[
\text{ITCH } = (\text{scratch} \rightarrow \text{STOP}).
\]
\[
\text{CONVERSE } = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}).
\]
\[
||\text{CONVERSE} \_\text{ITCH } = (\text{ITCH } || \text{ CONVERSE}).
\]

Possible traces as a result of action interleaving:

- think \(\rightarrow\) talk \(\rightarrow\) scratch
- think \(\rightarrow\) scratch \(\rightarrow\) talk
- scratch \(\rightarrow\) think \(\rightarrow\) talk
parallel composition - action interleaving

ITCH

2 states

CONVERSE

3 states

CONVERSEITCH

2 x 3 states

from ITCH

from CONVERSE

(0,0) (0,1) (0,2) (1,2) (1,1) (1,0)
parallel composition - algebraic laws

Commutative: \((P || Q) = (Q || P)\)
Associative: \((P || (Q || R)) = ((P || Q) || R) = (P || Q || R)\).

Clock radio example:

\[
\begin{align*}
\text{CLOCK} &= (\text{tick} \rightarrow \text{CLOCK}). \\
\text{RADIO} &= (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}). \\
\text{||CLOCK\_RADIO} &= (\text{CLOCK} || \text{RADIO}).
\end{align*}
\]

LTS? Traces? Number of states?
If processes in a composition have actions in common, these actions are said to be *shared*. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

**MAKER** = (make->ready->MAKER).
**USER** = (ready->use->USER).

||MAKER_USER = (MAKER || USER).

**MAKER** synchronizes with **USER** when **ready**.

*LTS? Traces? Number of states?*
modeling interaction - handshake

A handshake is an action acknowledged by another:

\[
\text{MAKERv2} = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKERv2}).
\]
\[
\text{USERv2} = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USERv2}).
\]
\[
| | \text{MAKER\_USERv2} = (\text{MAKERv2} \mid \mid \text{USERv2}).
\]

Interaction constrains the overall behaviour.
Three persons \( p[1..3] \) use a shared Russian bath, \textit{banya}. 

\begin{verbatim}
component SANDUNOVSKY
  {inst p[1..3] : PERSON;
  banya: BATH(3);
  bind p[1..3].enter -- banya.enter;
  bind p[1..3].exit -- banya.exit; }
\end{verbatim}
| | SANDUNOVSKY = ( p[1..Max]:PERSON
| | banya:BATH(3)
)\{p[1..Max].enter/ banya.enter,
p[1..Max].exit/ banya.exit\}.

SANDUNOVSKY


enter exit

banya: BATH(3)

instantiation
|| composition
/ relabelling
Composite Component – summary of FSP static operators

- Component composition is modelled as parallel composition \( \parallel \).
  \((\text{Interleaving of all the actions})\)

- Binding is modelled by relabelling \( / \).
  \((\text{Processes synchronise on shared actions})\)

- Composition expressions are direct translations from architecture descriptions.
Chapter 4. Behaviour analysis
Reachability analysis for checking models

Searches the *entire* system state space for **deadlock** states and **ERROR** states arising from property violations.

**Deadlock** - state with no outgoing transitions.

**ERROR** ($\pi$) state -1 is a trap state. Undefined transitions are automatically mapped to the **ERROR** state.
Safety - property automata

Safety properties are specified by deterministic finite state processes called property automata. These generate an image automata which is transparent for valid behaviour, but transitions to an ERROR state otherwise.

```plaintext
property EXCLUSION = ( p[i:1..3].enter
                        -> p[i].exit
                        -> EXCLUSION ).

||CHECK = (SANDUNOVSKY || EXCLUSION).
```

Safety properties are composed with the (sub)systems to which they apply, then check if ERROR is reachable in the composed system.

...if the number of spaces in the bath is 1? ...or 0?
Liveness - progress properties

To avoid the need to know LTL (Linear Temporal Logic), we directly support a limited class of liveness properties, called progress, which can be checked efficiently:

\[ \lozenge a \]

i.e. Progress properties check that, in an infinite execution, particular actions occur infinitely often.

For example:

\[
\text{progress OKtoBATH}[i:1..3] = \{p[i].enter\}
\]

...if we give priority to two of the bathers?
Compositional Reachability Analysis

We construct the system incrementally from subcomponents, based on the software architecture. State reduction is achieved by hiding actions not in their interfaces and minimising. Property checks remain in the minimised subcomponents.

Scalability

The problem with reachability analysis is that the state space “explodes” exponentially with increasing problem size.

How do we hope to alleviate this problem?

- **Compositional Reachability Analysis**
- **Partial Order Reduction**

As in SPIN, we employ on-the-fly analysis, exploring only that part of the state space which affects visible actions (cf. properties in SPIN). This can be done while preserving observational equivalence.
Chapter 5. Implementation in Java
Translation to Java

Identify active components (threads) & passive components (monitors):

FSP: \texttt{when \textit{cond} \textit{act} \rightarrow \textit{NEWSTAT}}

Java: \texttt{public synchronized void act() \{ \}

\begin{verbatim}
    \texttt{while (!cond) wait();}
    \texttt{// modify monitor data}
    \texttt{notifyAll()}
\end{verbatim}

\texttt{\}}
class Bath

class Bath {
    protected int spaces;
    protected int max;

    Bath(int n)
        {max = spaces = n;}

    synchronized void enter() throws InterruptedException {
        while (spaces==0) wait();
        --spaces;
        notifyAll();  //omit?
    }

    synchronized void exit() throws InterruptedException {
        while (spaces==n) wait();  //omit?
        ++spaces;
        notifyAll();
    }
}
class Person implements Runnable {
    Bath bath;

    Person(Bath b) {bath = b;}

    public void run() {
        try {
            while (true) {
                ... 
                bath.enter();
                <bathe actions>
                bath.exit();
                ...
            }
        } catch (InterruptedException e) {} 
    }
}
Chapter 6.  Graphical Animation – some examples

FMC  
NATS  
Chan  
Puzzle

LTS

\[
\begin{array}{c}
\text{a} \\
\text{b}
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b}
\end{array}
\]

\[
\begin{array}{c}
\text{x}
\end{array}
\]
Model analysis & animation

LTS Model checking
- safety properties
- progress properties
- compositional reachability
- abstraction & minimisation

Separate graphic animation model which preserves the behaviour of the model and has sound semantics based on Timed Automata.
A simple example - CHAN

CHAN = (in -> out -> CHAN
  | in -> fail -> CHAN
  ).
Models & Annotated models

Safety Properties
The annotated model cannot exhibit behavior that is not contained in the base model:

*Any safety property that holds for the base model also holds for the animated model.*
The animated model can thus be used to help understand the meaning of *counterexamples*.
Animated models can be composed to form complex models.
A simple workflow system – *OpenFlow*

Composite Task
NATS – short term conflict alert (STCA)

For each pair of aircraft determine potential conflict.

We can construct hybrid models that combine the discrete behavioural model with a real valued data stream.
Chapter 7. Logical Properties - states Vs events

Linear Temporal Logic

LTS

\[ a \xrightarrow{b} \xrightarrow{a} \xrightarrow{b} \]

\[ x \]
Properties – some deficiencies

- For simple models, safety properties are very similar to the model itself.

- Cannot specify some common liveness properties directly.
  
  e.g. \textit{Response} \texttt{[][}(request \rightarrow <> \texttt{reply})

Use the \texttt{Fluent Linear Temporal Logic model checker in LTSA tool} :-
Fluents - from the Event Calculus

"Fluents - time varying properties of the world.
Fluents are true at particular time-points if they have been initiated by an action occurrence at some earlier time-point and not terminated by another action occurrence in the meantime."

Miller & Shanahan
Fluent Propositions

Defined in terms of sets actions

fluent
LIGHT = \{on\}, \{power_cut,off\} initially False

[Magee & Giannakopoulou]
Fluents and the LTSA

- LTSA supports model checking of Fluent Linear Temporal Logic (FLTL)
  - Fluents
  - and (&&), or (||), implies (\rightarrow), not (\neg)
  - always ([]), eventually (<>), until (U),
  - weak until (W), next (X),
Using Fluents in SANDUNOVSKY

**fluent** BATHING\[i:1..Max]\n
\[= \langle p[i].enter, p[i].exit \rangle\]

//safety property

**assert** EXCLUSIONf = \[\!\!(\text{exists}[i:1..Max-1]

(BATHING[i] && BATHING [i+1..Max]))\]

//liveness property

**assert** OKtoBATHf = \forall[i:1..Max]

\[\exists\langle p[i].enter \rangle\]
Chapter 8. Dynamic and Adaptive Systems