Model-based design and analysis of concurrent and adaptive software

Jeff Kramer
Imperial College London

Models for concurrent software

Engineering is based on the use of simpler, abstract models for experimentation, reasoning and exhaustive analysis.

Abstraction? definitions ...

- the act of withdrawing or removing something
- the act or process of leaving out of consideration one or more properties of a complex object so as to attend to others

⇒ Remove detail (simplify) and focus (selection based on purpose)

- a general concept formed by extracting common features from specific examples
- the process of formulating general concepts by abstracting common properties of instances

⇒ generalisation (core or essence)

1930 – London Underground map

“Fit for purpose?”

Relationship between stations and interchanges, not actual distances
1932 – Harry Beck (1st schematic image map)

2001 – Fit for purpose (“mind the gap…”)

Fit for purpose ?!

“Underskin” by Samantha Loman

Fit for purpose – internationally!
Why is abstraction important in Software Engineering?

Software is abstract!

“Once you realize that computing is all about constructing, manipulating, and reasoning about abstractions, it becomes clear that an important prerequisite for writing (good) computer programs is the ability to handle abstractions in a precise manner.”

Keith Devlin CACM Sept.2003

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

Background: Book

Concurrency: State Models & Java Programs
Jeff Magee & Jeff Kramer
WILEY
2006 (2nd edition)
Background: Web based course material

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

structural view - Darwin ADL (Architecture Description Language)

- 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.

- Distributable context independent components with interaction via well defined interfaces
- An explicit configuration description (ADL) in terms of instantiation and binding
**Koala experience**

- "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**
- Component implementations
- Component behaviour models

**Construction View**
- Implementation

**Behavioural View**
- Modelling and Analysis

---

**Chapter 2. Modelling processes**

**Primitive components**
processes and threads

**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

---

**FSP – finite state processes**

**Component/Process:**

```java
component DRINKS {
    provides red; blue;
    requires tea; coffee;
}

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

---

**A countdown timer**

A countdown timer which beeps after N ticks, or can be stopped.

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

---

**FSP - guarded actions**

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

---

**FSP to model behaviour of the drinks machine:**

LTS:

```
DRINKS

0

1

2

3
```

---

**Java Demo**
A countdown timer

\[
\text{COUNTDOWN (N=3)} = (\text{start} \to \text{COUNTDOWN}[N]),
\]
\[
\text{COUNTDOWN}[i:0..N] =
\begin{cases}
  \text{tick} \to \text{COUNTDOWN}[i-1] & \text{when}(i>0) \\
  \text{beep} \to \text{STOP} & \text{when}(i==0) \\
  \text{STOP} & \text{stop}
\end{cases}
\]

\text{COUNTDOWN} = \{\text{start, tick, stop}\}.

\[
\text{PERSON} = (\text{enter} \to \text{bathe} \to \text{exit} \to \text{PERSON}) \oplus \{\text{enter, exit}\}.
\]

\text{PERSON can be minimised with respect to Milner's observational equivalence.}

\text{component PERSON - behaviour}

\text{component PERSON - behaviour}

\text{component PERSON - behaviour}

\text{component BATH - behaviour}

\text{component BATH (N=Max)}

\text{const Max = 3}

\text{range Int = 0..Max}

\text{BATH (N=Max) = BATH [N],}

\text{BATH[v:Int] = ( when(v>0) enter \to BATH[v-1]}

\text{| when(v<N) exit \to BATH[v+1] )}.

\text{PERSON enter exit}

\text{BATH enter exit enter exit enter exit}
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 $\rightarrow$
  - (guarded) choice $\mid$
  - recursion
- Interface $\Theta$ represents an action (or set of actions) in which the component can engage (i.e., 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)

parallel composition - **action interleaving**

If P and Q are processes then \( P || Q \) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

\[
ITCH = (\text{scratch} \rightarrow \text{STOP}).
\]

\[
CONVERSE = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}).
\]

\[
| | CONVERSE_ITCH = (ITCH || CONVERSE).
\]

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 - **algebraic laws**

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

Clock radio example:

\[
CLOCK = (\text{tick} \rightarrow \text{CLOCK}).
\]

\[
RADIO = (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}).
\]

\[
| | CLOCK_RADIO = (CLOCK || RADIO).
\]

LTS? Traces? Number of states?
**modeling interaction - shared actions**

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.

\[
\text{MAKER} = \text{(make} \rightarrow \text{ready} \rightarrow \text{MAKER)}. \\
\text{USER} = \text{(ready} \rightarrow \text{use} \rightarrow \text{USER)}. \\
\text{||MAKER\_USER} = (\text{MAKER} \parallel \text{USER}).
\]

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

**LTS? Traces? Number of states?**

Non-disjoint alphabets

**modeling interaction - handshake**

A handshake is an action acknowledged by another:

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

**Composite component behaviour**

**Component ROMA**

\[
\text{ROMA} = \{ \text{inst} p[1..3] : PERSON; \\
\text{bagno} : \text{BATH(3)}; \\
\text{bind} p[1..3].\text{enter} \rightarrow \text{bagno.}\text{enter}; \\
\text{bind} p[1..3].\text{exit} \rightarrow \text{bagno.}\text{exit}; \}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>enter</td>
<td>exit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bagno: BATH(3)</td>
</tr>
</tbody>
</table>

**Composite component behaviour - FSP**

\[
\text{||ROMA} = \{ \text{p[1..Max] : PERSON} \\
\text{|| bagno} : \text{BATH(Max)} \\
\} / \{ \text{p[1..Max].}\text{enter} / \text{bagno.}\text{enter}, \\
\text{p[1..Max].}\text{exit} / \text{bagno.}\text{exit} \}.
\]

**Three persons** p[1..3] use a shared Roman bath, **bagno**.
Composite component behaviour - FSP

\[ \text{||ROMA} = (\ p[1..\text{Max}]:\text{PERSON} \ \ \ \ \ ||\ p[1..\text{Max}]:\text{BATH}(\text{Max}) ) \] .

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 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.

\[
\text{property EXCLUSION = ( } p[i:1..3].\text{enter} \\
\text{ } \rightarrow p[i].\text{exit} \\
\text{ } \rightarrow \text{EXCLUSION }) .
\]

\[||\text{CHECK = (ROMA}|| \text{EXCLUSION)}.\]

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?

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:** when cond act -> NEWSTAT

**Java:**
```java
public synchronized void act()
    throws InterruptedException
{
    while (!cond) wait();
    // modify monitor data
    notifyAll();
}
```

**Person threads**
```java
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**

**class Bath**
```java
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();
    }
    synchronized void exit() throws InterruptedException {
        while (spaces==n) wait();
        ++spaces;
        notifyAll();
    }
}
```
Model analysis & animation

LTS model

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.

Puzzle

The animated model can thus be used to help understand the meaning of counterexamples.
Flexible Manufacturing Cell

Animated models can be composed to form complex models.

A simple workflow system – OpenFlow

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
Properties – some deficiencies

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

- Cannot specify some common liveness properties directly
  e.g. Response \([\text{request} \rightarrow \leftrightarrow \text{reply}]\)

Use the Fluent Linear Temporal Logic model checker in LTSA tool:

Defining Abstract States over Sequences of Events

◆ 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

Fluents and the LTSA

◆ LTSA supports model checking of Fluent Linear Temporal Logic (FLTL)
  - Fluents
  - and (\&\&), or (||), implies (\rightarrow), not (!)
  - always ([]), eventually (\leftrightarrow), until (U),
  - weak until (W), next (X),

Fluent Propositions

Defined in terms of sets actions

\[
\begin{align*}
\text{fluent} & \quad \text{LIGHT} = \langle \{\text{on}\}, \{\text{power_cut,off}\} \rangle \\
& \quad \text{initially False}
\end{align*}
\]

[Magee & Giannakopoulou]
Using Fluents in ELBA

**fluent** BATHING[i:1..Max] = <p[i].enter,p[i].exit>

//safety property
**assert** EXCLUSIONf = []!(exists[i:1..Max-1] (BATHING[i] && BATHING [i+1..Max]))

//liveness property
**assert** OKtoBATHf = forall[i:1..Max] []<>p[i].enter

---

La Scala Revision example

---

Revision Example: A Concert Hall Booking System

A central computer connected to remote terminals via communication links is used to automate seat reservations for a concert hall.

To book a seat, a client chooses a free seat and the clerk enters the number of the chosen seat at the terminal and issues a ticket, if it is free.

A system is required which avoids double bookings of the same seat whilst allowing clients free choice of the available seats.

*Construct an abstract model of the system and demonstrate that your model does not permit double bookings.*

---

Concert Hall Booking System

```
const False = 0
const True  = 1
range Bool = False..True

SEAT = SEAT[False],
SEAT[reserved:Bool] = ( when (!reserved) reserve -> SEAT[True]
| query[reserved] -> SEAT[reserved]
| when (reserved) reserve -> ERROR //error of reserved twice
).

range Seats = 1..2
||SEATS = (seat[Seats]:SEAT).
```

ERROR is the trap state, numbered -1 in the equivalent LTS.
Concert Hall Booking System

TERMINAL = (choose[s:Seats] -> seat[s].query[reserved:Bool] -> if (!reserved) then (seat[s].reserve -> TERMINAL) else TERMINAL).

set Terminals = {a,b}

||CONCERT = ( Terminals:TERMINAL || Terminals::SEATS ).

Does this allow double booking of a seat?

Concert Hall Booking System – no interference?

LOCK = (acquire -> release -> LOCK).

//lock for the booking system

TERMINAL = (choose[s:Seats] -> acquire
-> seat[s].query[reserved:Bool] -> if (!reserved) then (seat[s].reserve -> release -> TERMINAL) else (release -> TERMINAL) ).

set Terminals = {a,b}

||CONCERT = (Terminals:TERMINAL || Terminals::SEATS || Terminals::LOCK).

Would locking at the seat level permit more concurrency?

Concert Hall Booking System – locking at seat level

LOCK = (acquire -> release -> LOCK).

//lock for the booking system

||LOCKS = (lock[Seats]:LOCK). //lock for each seat

TERMINAL = (choose[s:Seats] -> lock[s].acquire
-> seat[s].query[reserved:Bool] -> if (!reserved) then (seat[s].reserve->lock[s].release->TERMINAL) else (lock[s].release -> TERMINAL) ).

set Terminals = {a,b}

||CONCERT = (Terminals:TERMINAL || Terminals::SEATS || Terminals::LOCKS).

Chapter 8. Dynamic and Adaptive Systems