Software Architecture

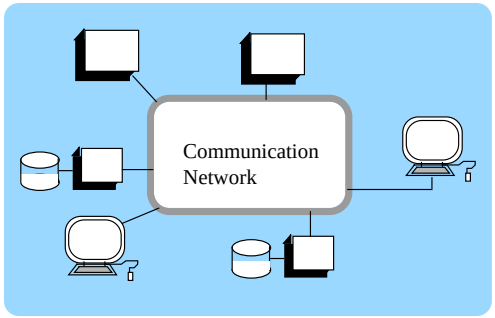
Modeling & Analysis:

a rigorous approach

Jeff Kramer & Jeff Magee

Imperial College, London.

concurrent and distributed software components?



Interacting, concurrent software components of a system:

single machine -> *shared memory interactions*

multiple machines -> *network interactions*

System is a composition of concurrent software components according to the software architecture.

Do I need to know about concurrent architectures?

Concurrency is widespread but error prone.

- Therac - 25 computerised radiation therapy machine

  Concurrent programming errors contributed to accidents causing deaths and serious injuries.

- Mars Rover

  Problems with interaction between concurrent tasks caused periodic software resets reducing availability for exploration.

a Cruise Control System

When the car ignition is switched on and the **on** button is pressed, the current speed is recorded and the system is enabled: *it maintains the speed of the car at the recorded setting.*

Pressing the brake, accelerator or **off** button disables the system. Pressing **resume** re-enables the system.

- ***Is the system safe?***
- ***Would testing be sufficient to discover all errors?***

models

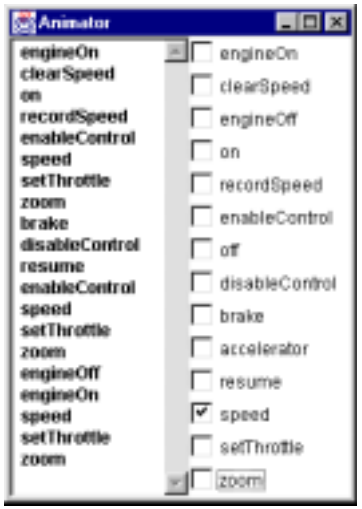
A model is a simplified representation of the real world.

Engineers use models to gain confidence in the adequacy and validity of a proposed design.

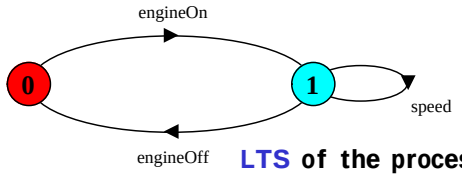
- focus on aspect of interest – concurrency & composition
- model animation to visualise a behaviour
- mechanical verification of properties (safety & progress)

Our models are described using state machines, known as Labelled Transition Systems **LTS**. These are described textually as finite state processes (**FSP**) and displayed and analysed by the ***LTSA*** analysis tool.

modelling the Cruise Control System



LTSA **Animator to step through system actions and events.**



LTS of the process that monitors speed.

Later we will explain how to construct models such as this so as to perform animation and verification.

tutorial objective

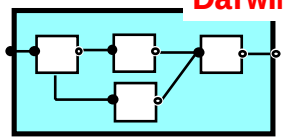
This tutorial is intended to provide an introduction into **model-based design** of concurrent software, where the proposed architecture provides the structure.

We illustrate how models can be used to provide insight into behavior and to aid reasoning about particular designs.

Furthermore, we investigate how requirements scenarios can be used to help construct models.

Model based approach

Software Architecture describes gross organization of a system in terms of components and their interactions.

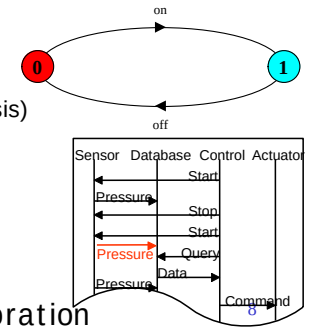


Behavior Modelling:

State Machines in form of LTS (Labelled Transition Systems)

Analysis using Model Checking CRA (Compositional Reachability Analysis) and LTL (Linear Temporal Logic)

Interpretation using animation



Requirements in the form of MSC (Message Sequence Charts) for model generation and elaboration

Background: Web based material

http://www-dse.doc.ic.ac.uk/concurrency/

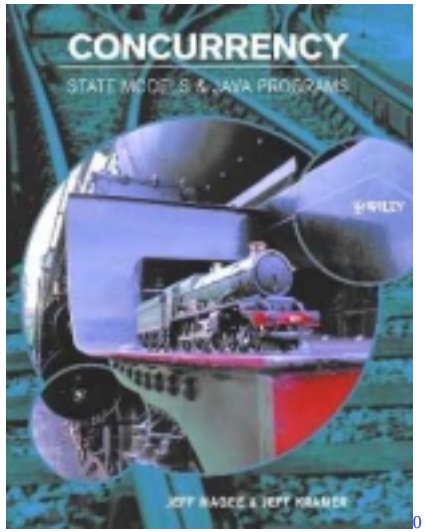
- Java examples and demonstration programs
- State models for the examples
- Labelled Transition System Analyser (***LTSA***) for modeling concurrency, model animation and model property checking.
- Plugins for the Darwin architecture and MSC scenario descriptions.

Background: Book

Concurrency: State Models & Java Programs

Jeff Magee & Jeff Kramer

WILEY



Tutorial Structure

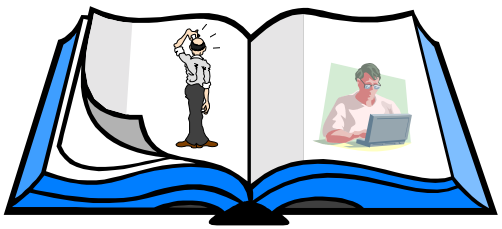
A: Models and Analysis

B: Architectural Description and Models

C: Animation and Analysis

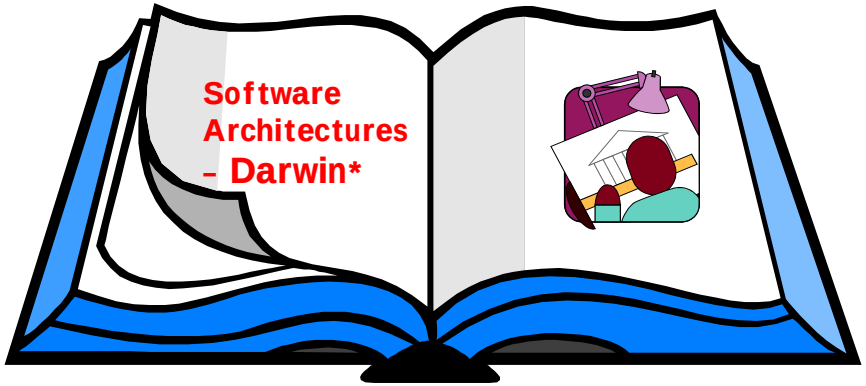
D: Models from Scenarios

E: Conclusion



Part A :
Models and Analysis

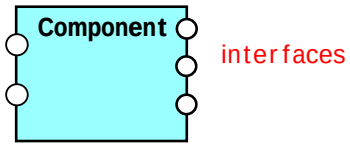
1. Software structure



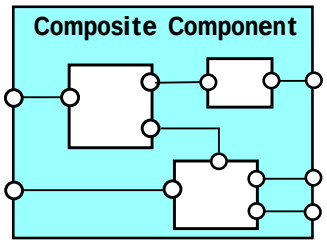
**Darwin is described in a later section*

Software structure – composition of components*

- Components have one or more **interfaces**. An interface is simply a set of names referring to actions in a specification or functions in an implementation.

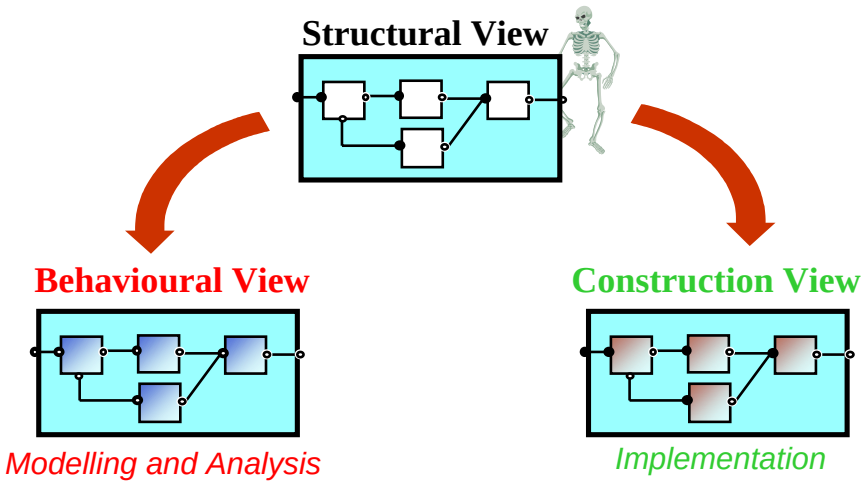


- Systems / composite components are composed hierarchically by component **instantiation** and interface **binding**.

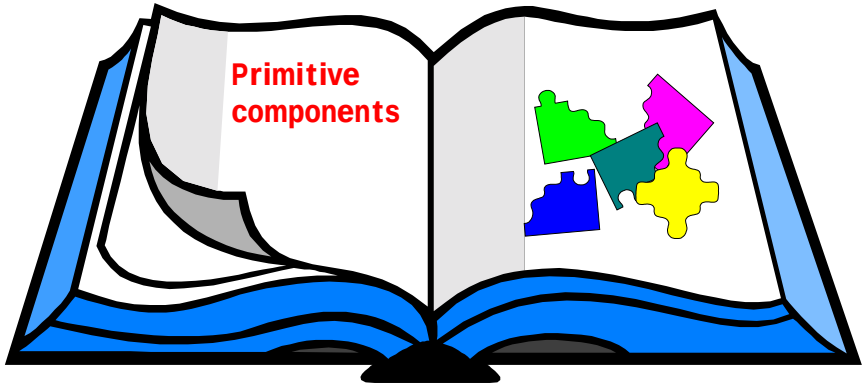


**based on Darwin, an architectural description language (ADL)*

Architectural description - multiple views



2. Modelling: Processes and Threads



processes and threads

Concepts: 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**

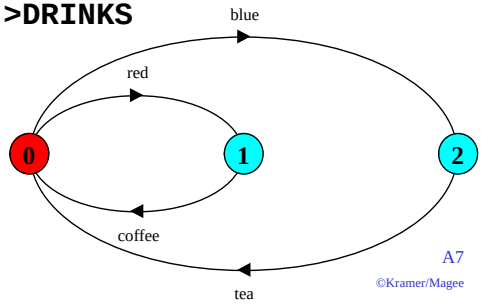
FSP – finite state processes

Component/Process:

FSP to model behaviour of the drinks machine :

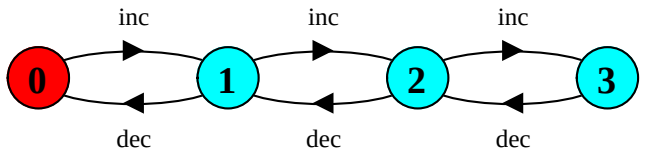
```
DRINKS = (red->coffee->DRINKS
         |blue->tea->DRINKS
         ).
```

LTS generated using *LTSA*:



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]
                ).
```



A countdown timer

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

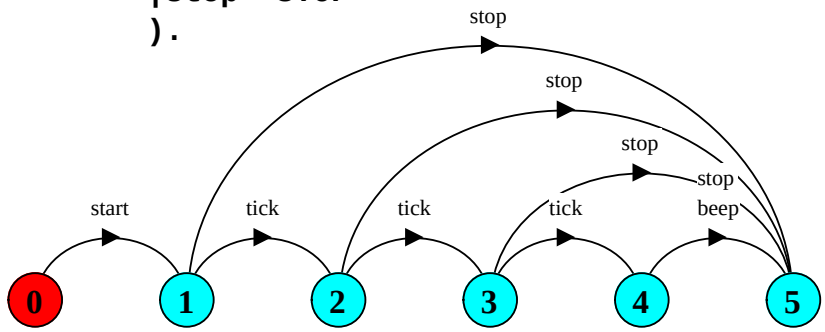
Java Demo

FSP?

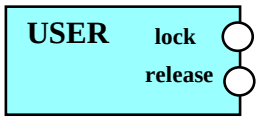
LTS?

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
        ).
```



USER of a resource

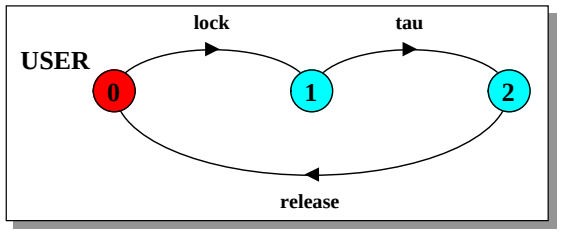


Process specification FSP:

```
USER = (lock -> critical -> release -> USER
       )@{lock,release}.
```

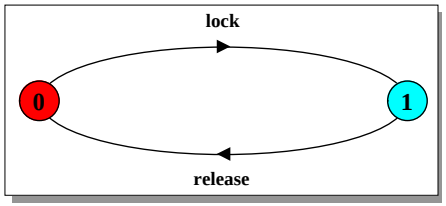

USER

Labelled transition system LTS:

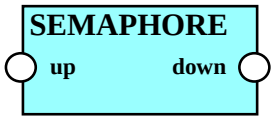


Animation can be used to step through the actions to test specific scenarios.

USER can be minimised with respect to Milner's observational equivalence.

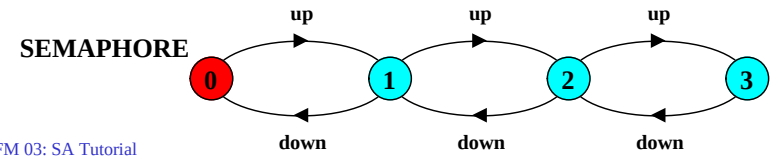


SEMAPHORE - behaviour

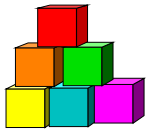


Process specification FSP:

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int]    = (when(v<Max) up -> SEMA[v+1]
                 |when(v>0) down -> SEMA[v-1]
                 ).
```

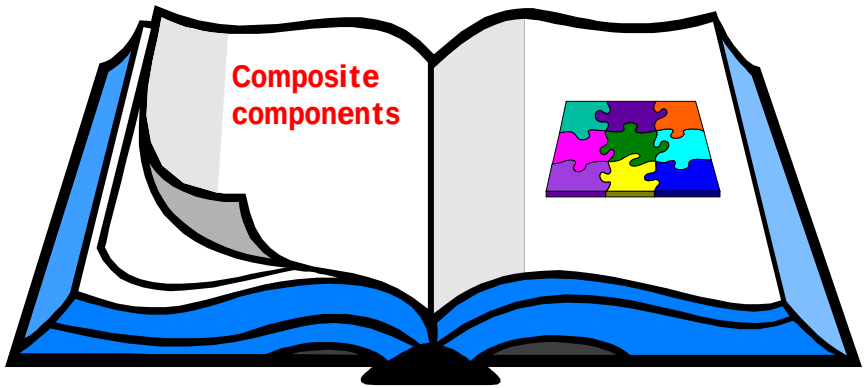


Primitive Components - summary



- Component behaviour is modelled using Labelled Transition Systems (LTS).
- Primitive components are described as finite state processes using:
  - **action prefix ->**
  - **choice | (guarded)**
  - **recursion**
  - **interface @**
- interface represents an action (or set of actions) in which the component can engage.

3. Modelling – concurrent processes and threads



Concurrent execution

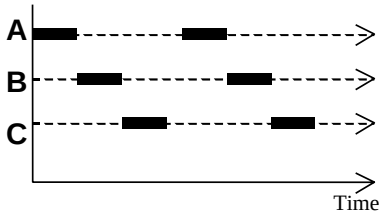
Concepts: processes - concurrent execution
and interleaving.
process interaction.

Models: **parallel composition** of asynchronous processes
- interleaving
interaction - shared actions
process labeling, and action relabeling and hiding
structure diagrams

Practice: Multithreaded Java programs

Definitions

- ***Concurrency***
  - *Logically* simultaneous processing. Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.
- ***Parallelism***
  - *Physically* simultaneous processing. Involves multiple PEs and/or independent device operations.



Both concurrency and parallelism require controlled access to shared resources . We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-concurrent execution.

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

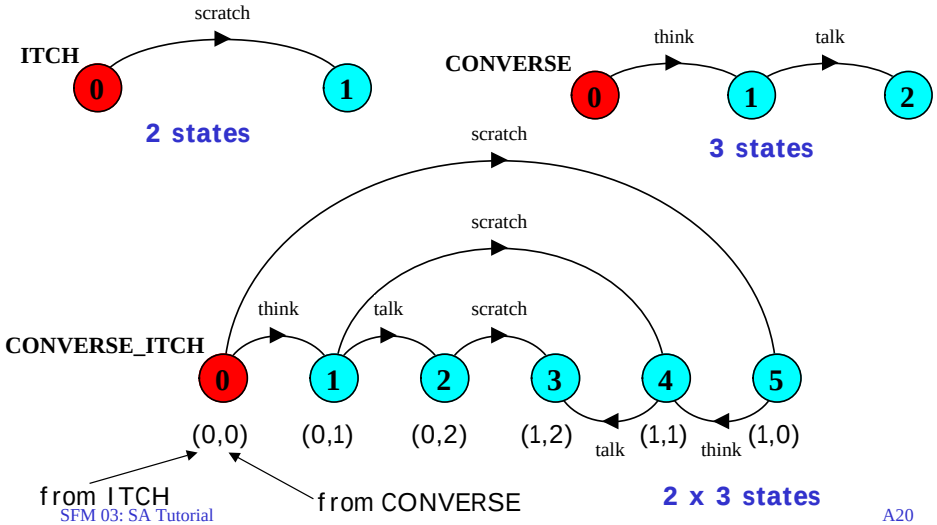
If P and Q are processes then (P||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

```
ITCH  = (scratch->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSE_ITCH = (ITCH || CONVERSE).
```

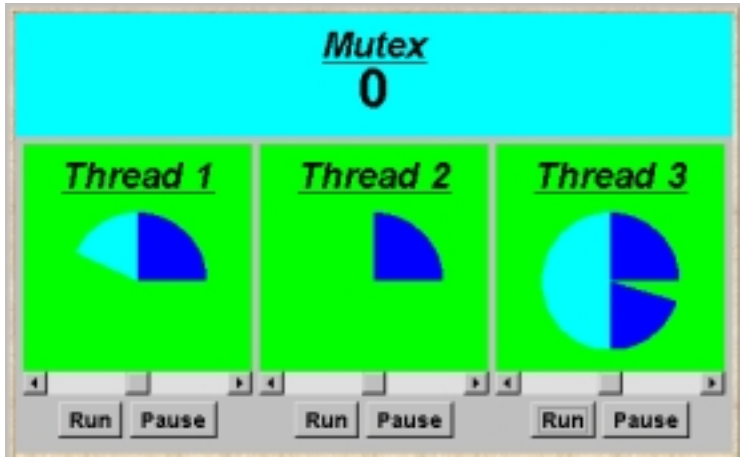
think→talk→scratch
think→scratch→talk
scratch→think→talk

Possible traces as a result of action interleaving.

parallel composition - action interleaving



SEMADEMO



current semaphore value

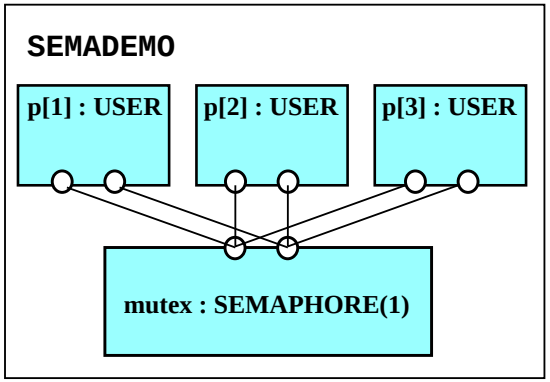
thread 1 is executing critical actions.

thread 2 is blocked waiting.

thread 3 is executing non-critical actions.

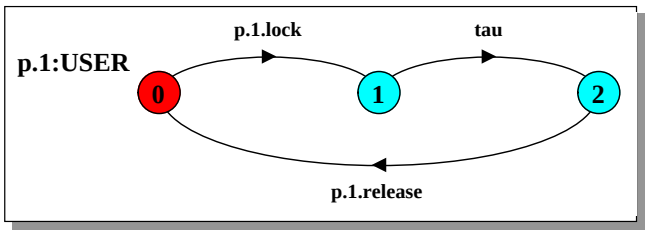
Composite component behaviour - interaction

Three processes **p[1..3]** use a shared semaphore **mutex** to ensure mutually exclusive access to some resource.



Composite component behaviour

```
||SEMADEMO = (p[1..3]:USER
            || mutex:SEMAPHORE(1)
            ) /{p[1..3].lock/mutex.down,
                p[1..3].release/mutex.up}.
```



:	instantiation
||	composition
/	relabelling

For mutual exclusion, the initial semaphore value must be **1**.

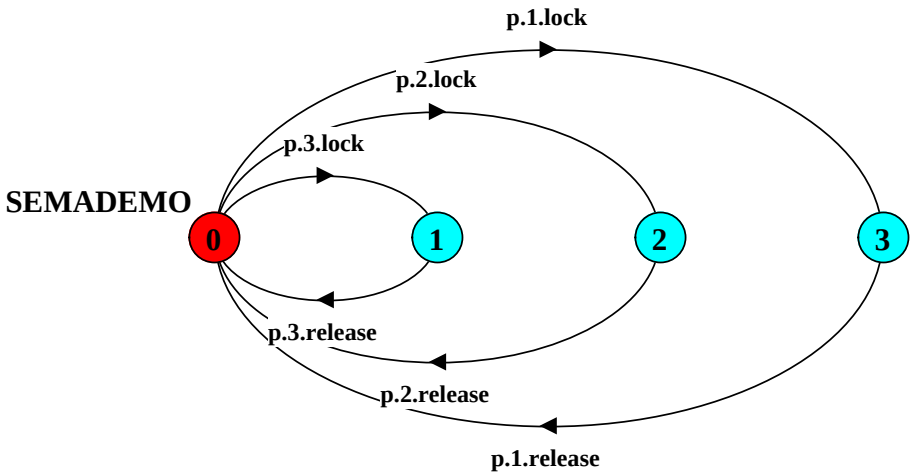
Parallel Composition with shared actions

Parallel composition **||** generates an LTS that represents all possible interleaving of the actions. Processes synchronise on shared actions.

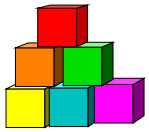
Minimise with respect to hidden actions (not in interface).

```
SEMADEMO = p.1:USER || p.2:USER || p.3:USER
|| mutex:SEMAPHORE(1)
State Space:
 3 * 3 * 3 * 4 = 108
Composing
States Composed: 7 Transitions: 9 in 0ms
SEMADEMO minimising...
Minimised States: 4 in 0ms
```

SEMADEMO LTS

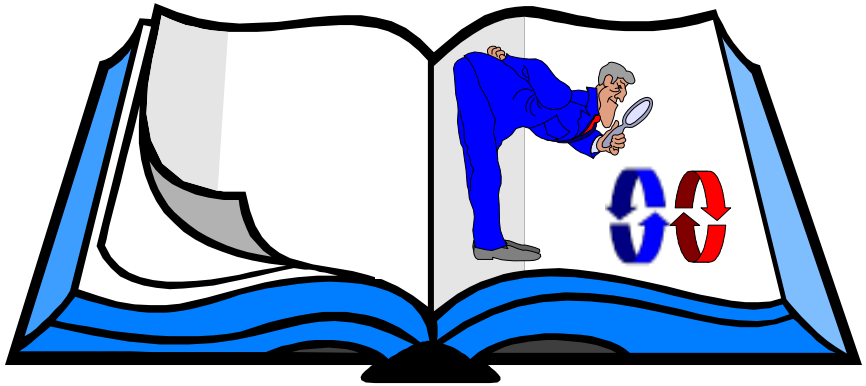


Composite Component - summary



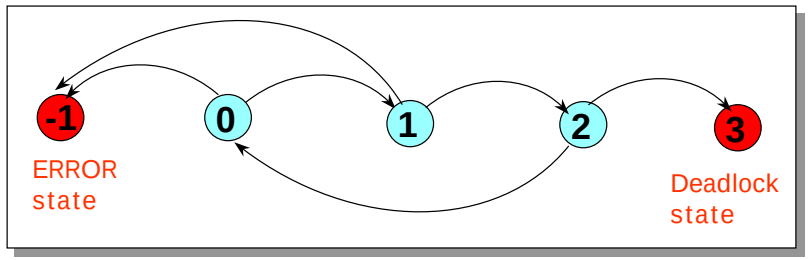
- Component composition is modelled as parallel composition **||**.
  (Interleaving of all the actions)
- Binding is modelled by relabelling ***/***.
  (Processes synchronise on shared actions)
- Composition expressions are **direct** translations from architecture descriptions.

4. Behaviour analysis – model checking



Reachability analysis for checking models

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



A deadlock is a state with no outgoing transitions.

The **ERROR** state -1 is a trap state. Undefined transitions are automatically mapped to the **ERROR** state.

Deadlock

Set initial semaphore value to 0. Result?

```
SEMADEMO = p.1:USER || p.2:USER || p.3:USER
 || mutex:SEMAPHORE(0)
State Space:
 3 * 3 * 3 * 4 = 108
Composing
 potential DEADLOCK
States Composed: 1 Transitions: 0 in 0ms
Trace to DEADLOCK:
```


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 (good) behaviour, but transitions to an **ERROR** state otherwise.

```
property EXCLUSION =
    (p[i:1..3].lock -> p[i].release
     -> EXCLUSION ).

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

Safety properties are composed with the (sub)systems to which they apply.

safety analysis

```
Composition:
CHECK = SEMADEMO.p.1:USER || SEMADEMO.p.2:USER ||
SEMADEMO.p.3:USER || SEMADEMO.mutex:SEMAPHORE(1)
|| EXCLUSION
State Space:
 3 * 3 * 3 * 4 * 4 = 432
Composing
States Composed: 7 Transitions: 9 in 0ms
No deadlocks/errors
```

Is **SEMADEMO** safe when the initial value of the semaphore is 2?

Liveness - progress properties

Support a limited class of liveness properties, called progress, that can be checked efficiently :

$$[]\Diamond a$$

$$[]\Diamond a \Rightarrow []\Diamond b$$

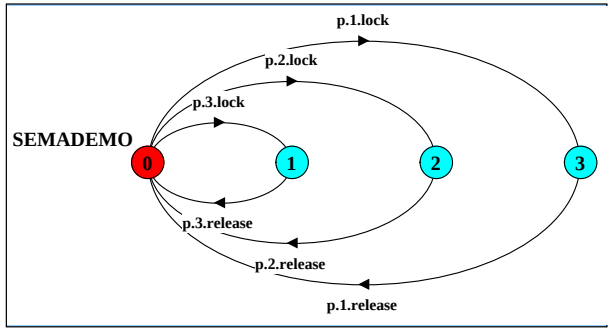
i.e. Progress properties check that, in an infinite execution, **particular actions** occur infinitely often.

For example:

```
progress AQUIRE[i:1..3] = {p[i].lock}
```

Fair Choice

If a choice over a set of transitions is executed infinitely often, then every transition in the set is executed infinitely often.



How do we check progress?

each terminal set of states should contain at least one of the actions required by the property.

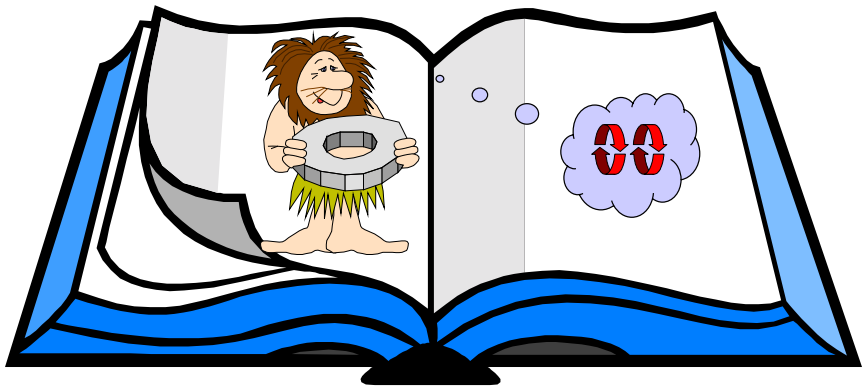
Action Priority

Action priority imposes adverse conditions:

high : PROCESS << {actions}
low : PROCESS >> {actions}

```
||PRIORITY1 = SEMADEMO << {p[1].lock}.

||UNFAIR1 = SEMADEMO >> {p[1].lock}.
```


5. An example of Model-based Design



Design

Concepts: **design process:**
requirements to **models** to implementations

Models: check properties of interest:
- safety on the appropriate (sub)system
- progress on the overall system

Practice: model interpretation - to infer actual system behaviour
threads and monitors

Aim: rigorous design process.

from requirements to models

Requirements

- goals of the system
- scenarios (Use Case models) *
- properties of interest

Any appropriate design approach can be used.

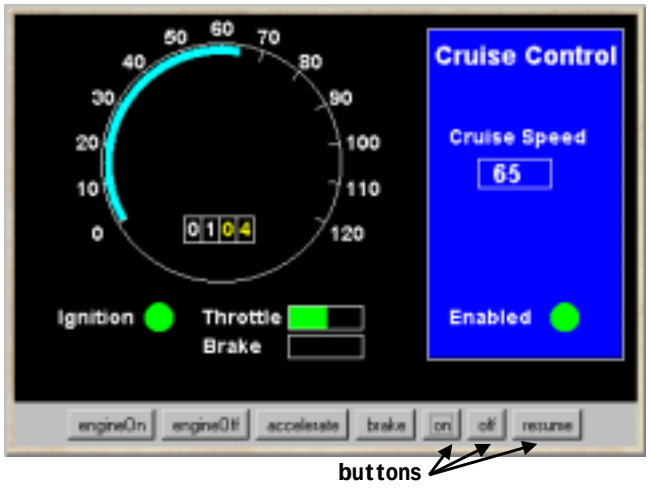
- identify the main events, actions, and interactions
- identify and define the main processes
- identify and define the properties of interest
- structure the processes into an **architecture**

Model

- check traces of interest
- check properties of interest

* *Generating and elaborating models from scenarios is described in a later section.*

a Cruise Control System - requirements



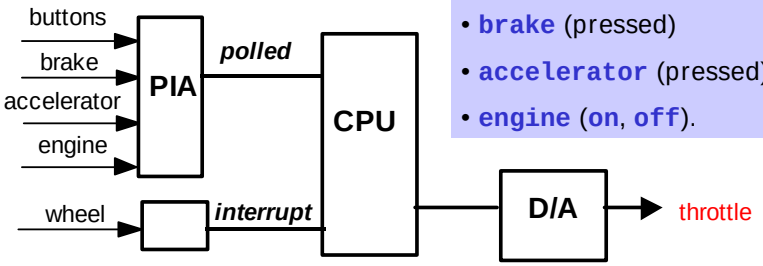
When the car ignition is switched on and the **on** button is pressed, the current speed is recorded and the system is enabled: *it maintains the speed of the car at the recorded setting*.

Pressing the brake, accelerator or **off** button disables the system. Pressing **resume** or on re-enables the system.

a Cruise Control System - hardware

Parallel Interface Adapter (PIA) is polled every 100msec. It records the actions of the sensors:

- buttons (**on**, **off**, **resume**)
- **brake** (pressed)
- **accelerator** (pressed)
- **engine** (**on**, **off**).

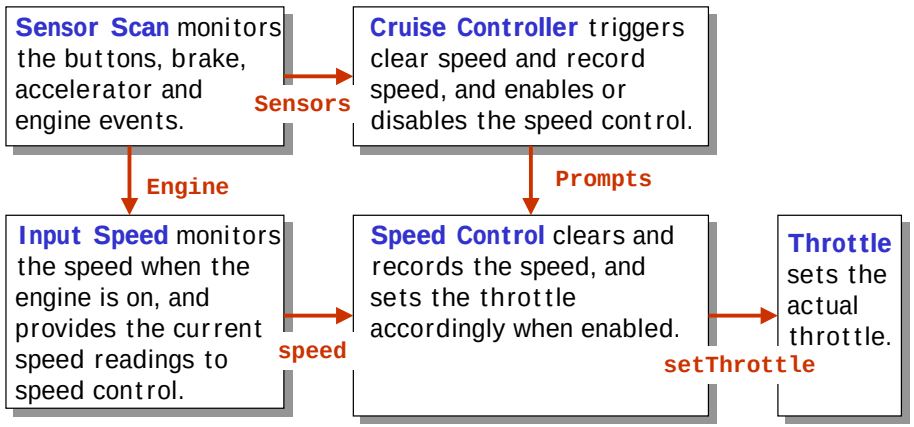


Wheel revolution sensor generates interrupts to enable the car **speed** to be calculated.

Output: The cruise control system controls the car speed by setting the **throttle** via the digital-to-analogue converter.

model - outline design

- outline processes and interactions.



model -design

- Main events, actions and interactions.

  Sensors: `on`, `off`, `resume`, `brake`, `accelerator`, `engine on`, `engine off`, `speed`, `setThrottle`

  Prompts: `clearSpeed`, `recordSpeed`, `enableControl`, `disableControl`

- Identify main processes.

  `Sensor Scan, Input Speed, Cruise Controller, Speed Control and Throttle`

- Identify main properties.

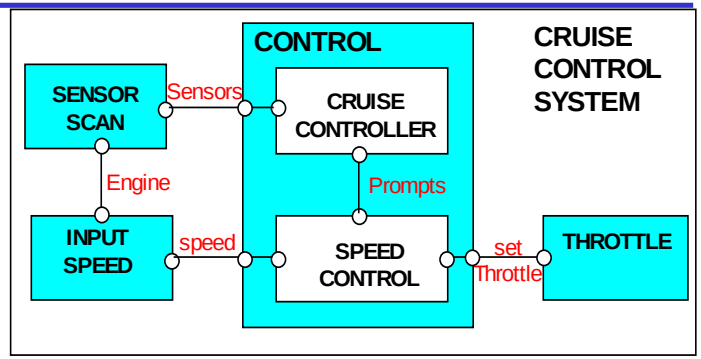
  `safety` - disabled when `off`, `brake` or `accelerator` pressed.

- Define and structure each process.

model - structure, actions and interactions

The CONTROL system is structured as two processes.

The main actions and interactions are as shown.




```
set Sensors = {engineOn,engineOff,on,off,
               resume,brake,accelerator}
set Engine  = {engineOn,engineOff}
set Prompts = {clearSpeed,recordSpeed,
               enableControl,disableControl}
```


model elaboration - process definitions

```
SENSORSCAN = ({Sensors} -> SENSORSCAN).
     // monitor speed when engine on
INPUTSPEED = (engineOn -> CHECKSPEED),
CHECKSPEED = (speed -> CHECKSPEED
             |engineOff -> INPUTSPEED
             ).
     // zoom when throttle set
THROTTLE =(setThrottle -> zoom -> THROTTLE).
     // perform speed control when enabled
SPEEDCONTROL = DISABLED,
DISABLED =({speed,clearSpeed,recordSpeed}->DISABLED
          | enableControl -> ENABLED
          ),
ENABLED = ( speed -> setThrottle -> ENABLED
          |{recordSpeed,enableControl} -> ENABLED
          | disableControl -> DISABLED
          ).
```


model elaboration - process definitions

```
     // enable speed control when cruising,
     // disable when off, brake or accelerator pressed
CRUISECONTROLLER = INACTIVE,
INACTIVE =(engineOn  -> clearSpeed -> ACTIVE),
ACTIVE   =(engineOff -> INACTIVE
          |on->recordSpeed->enableControl->CRUISING
          ),
CRUISING =(engineOff -> INACTIVE
          |{off,brake,accelerator}
                    -> disableControl -> STANDBY
          |on->recordSpeed->enableControl->CRUISING
          ),
STANDBY  =(engineOff -> INACTIVE
          |resume -> enableControl -> CRUISING
          |on->recordSpeed->enableControl->CRUISING
          |{off,brake,accelerator} -> STANDBY
          ).
```

model - CONTROL subsystem

```
||CONTROL =(CRUISECONTROLLER
           ||SPEEDCONTROL
           ).
```

Animate to check particular traces:

- Is control enabled after the engine is switched on and the on button is pressed?
- Is control disabled when the brake is then pressed?
- Is control re-enabled when resume is then pressed?

However, we need to analyse to exhaustively check:

Safety: Is the control disabled when **off**, **brake** or **accelerator** is pressed?
Progress: Can every action eventually be selected?

model - Safety properties

```
property CRUISESAFETY =
  ({off,accelerator,brake,disableControl} -> CRUISESAFETY
  |{on,resume} -> SAFETYCHECK
  ),
SAFETYCHECK =
  ({on,resume} -> SAFETYCHECK
  |{off,accelerator,brake} -> SAFETYACTION
  |disableControl -> CRUISESAFETY
  ),
SAFETYACTION =(disableControl->CRUISESAFETY).
```

LTS?

```
||CONTROL =(CRUISECONTROLLER
           ||SPEEDCONTROL
           ||CRUISESAFETY
           ).
```

Is **CRUISESAFETY** *violated?*

model analysis

We can now compose the whole system:

```
||CONTROL =
   (CRUISECONTROLLER||SPEEDCONTROL||CRUISESAFETY
    ).
||CRUISECONTROLSYSTEM =
    (CONTROL||SENSORSCAN||INPUTSPEED||THROTTLE).
```

Deadlock? Safety?

```
No deadlocks/errors
```

Progress?

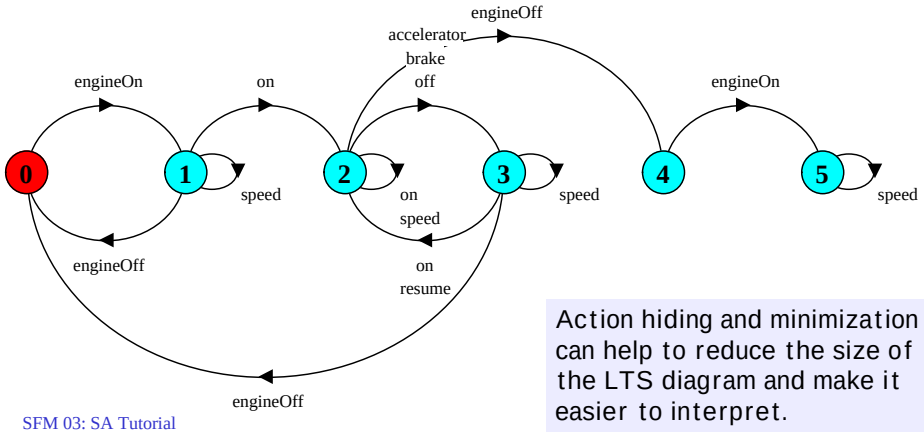
model - Progress properties

```
Progress violation for actions:
{engineOn, clearSpeed, engineOff, on, recordSpeed,
enableControl, off, disableControl, brake,
accelerator...........}
Path to terminal set of states:
     engineOn
     clearSpeed
     on
     recordSpeed
     enableControl
     engineOff
     engineOn
Actions in terminal set:
{speed, setThrottle, zoom}
```

Control is not disabled when the engine is switched off !

cruise control model - minimized LTS

```
||CRUISEMINIMIZED = (CRUISECONTROLSYSTEM)
                    @ {Sensors,speed}.
```



Action hiding and minimization can help to reduce the size of the LTS diagram and make it easier to interpret.

model - revised cruise control system

Modify **CRUISECONTROLLER** so that control is disabled when the engine is switched off:

```
…
CRUISING =(engineOff -> disableControl -> INACTIVE
          |{ off,brake,accelerator} -> disableControl -> STANDBY
          |on->recordSpeed->enableControl->CRUISING
          ),
…
```

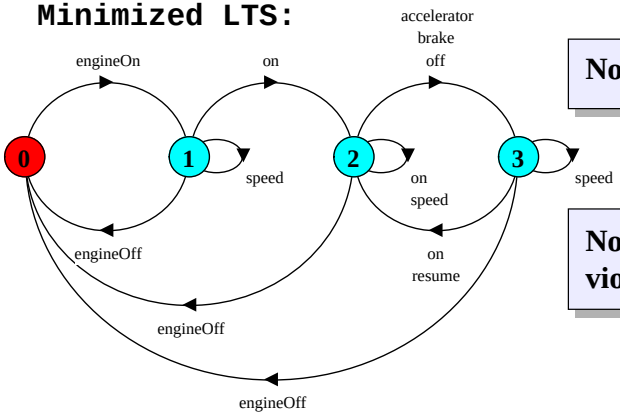
OK now?

Modify the safety property:

```
property IMPROVEDSAFETY = ({off,accelerator,brake,disableControl,
                                        engineOff} -> IMPROVEDSAFETY
                           |{on,resume}       -> SAFETYCHECK
                           ),
SAFETYCHECK = ({on,resume}  -> SAFETYCHECK
              |{off,accelerator,brake,engineOff} -> SAFETYACTION
              |disableControl    -> IMPROVEDSAFETY
              ),
SAFETYACTION =(disableControl     -> IMPROVEDSAFETY).
```

model - revised cruise control system

Minimized LTS:



No deadlocks/errors

No progress violations detected.

model interpretation

Models can be used to indicate system sensitivities.

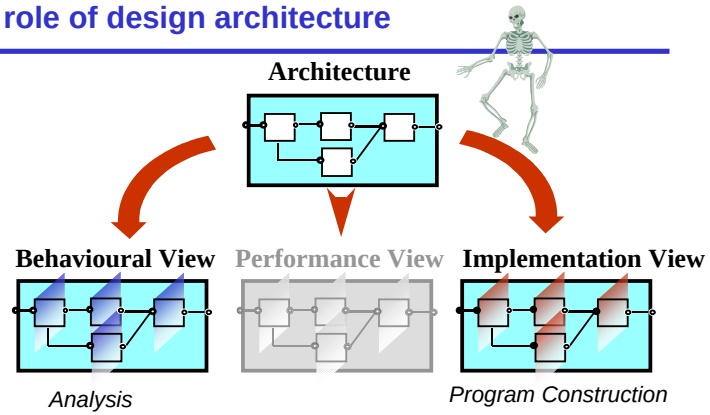
If it is possible that erroneous situations detected in the model may occur in the implemented system, then the model should be revised to find a design which ensures that those violations are avoided.

However, if it is considered that the real system will not exhibit this behaviour, then no further model revisions are necessary.

Model interpretation and correspondence to the implementation are important in determining the relevance and adequacy of the model design and its analysis.

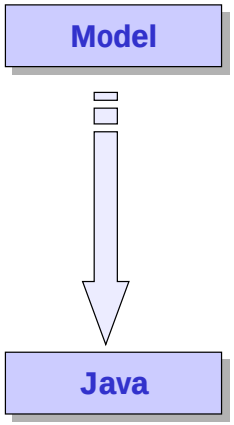
The central role of design architecture

Design architecture describes the gross organization and global structure of the system in terms of its constituent components.



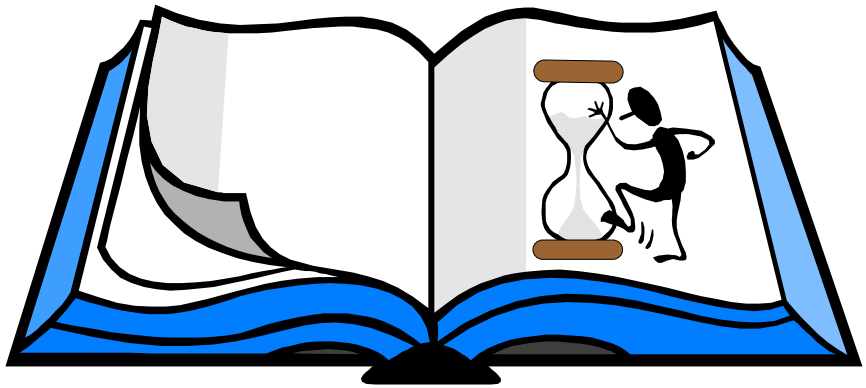
We consider that the models for analysis and the implementation should be considered as elaborated views of this basic design structure.

from models to implementations



- identify the main active entities
  - to be implemented as threads
- identify the main (shared) passive entities
  - to be implemented as monitors
- identify the interactive display environment
  - to be implemented as associated classes
- structure the classes as a class diagram

6. Summary ...



Summary

- Concepts
  - **design process:**
    from requirements to models to implementations
  - **design architecture**
- Models
  - **check properties of interest**
    safety: ***compose safety properties at appropriate (sub)system***
    progress: ***apply progress check on the final target system model***
- **Practice**
  - **model interpretation - to infer actual system behaviour**
  - **threads and monitors**

Aim: rigorous design process.

Related Work

- ADL Wright + FDR toolset
- LOTOS + Caesar/Aldebaran
- Promela + SPIN

Our approach is distinguished by:

- direct use of ADL to generate both analysis model & implementation,
- emphasis on compositionality.

Background: Web based material

http://www-dse.doc.ic.ac.uk/concurrency/

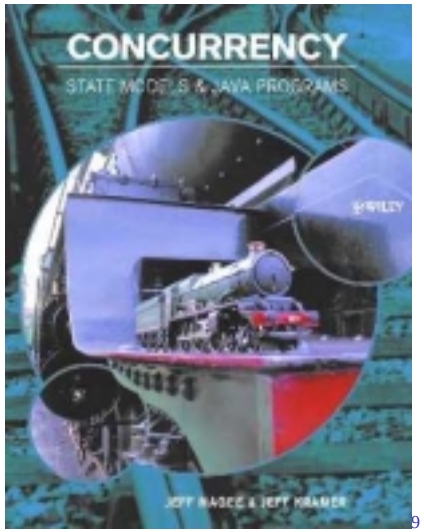
- Java examples and demonstration programs
- State models for the examples
- Labelled Transition System Analyser (***LTSA***) for modeling concurrency, model animation and model property checking.
- Plugins for the Darwin architecture and MSC scenario descriptions.

Background: Book

Concurrency: State Models & Java Programs

Jeff Magee & Jeff Kramer

WILEY



Current work ...

- Investigating extension of models to support other views:
  - property analysis using LTL and fluents
  - domain specific animation based on Timed Automata
  - performance based on Stochastic Process Algebra
  - model generation and elaboration from MSCs
  - web application animation from MSCs

Approach supported by the **Labelled Transition System Analyser (LTSA)** available from:

http://www-dse.doc.ic.ac.uk/concurrency/

Models + Analysis Further Reading

[1] Magee, J. and Kramer, J., "Concurrency: State Models & Java Programs", *John Wiley & Sons* (Worldwide Series in Computer Science), March 1999, 355 pages.

[2] Kramer, J., and Magee, J., "Exposing the Skeleton in the Coordination Closet", in *Coordination Languages and Models, 2nd International Conference COORDINATION '97*, Berlin, 1997, 18-31.

[3] Cheung, S.C., and Kramer, J., "Checking Safety Properties using Compositional Reachability Analysis", *ACM Transactions on Software Engineering Methodology TOSEM*, 8 (1), 1999, 49-78.

[4] Cheung, S.C., and Kramer, J., "Context Constraints for Compositional Reachability Analysis", *ACM Transactions on Software Engineering Methodology TOSEM*, 5 (4), (1996), 334-377.

[5] Giannakopoulou D., Magee J., and Kramer J., "Checking Progress with Action Priority: Is it Fair?", *7th ACM SIGSOFT Symposium on the Foundations of Software Engineering / 7th European Software Engineering Conference (FSE / ESEC '99)*, Toulouse, September 1999), LNCS , (Springer-Verlag), 1999, 511-528.

[6] Kramer, J., and Magee, J., "Modelling for mere Mortals", Keynote: *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, Amsterdam, March 1999, 1-18.

Part B :
Architectural Description & Models

The central role of design architecture

- Design architecture describes the gross organization and global structure of the system in terms of its constituent components.
- We consider that the models for analysis and the implementation should be considered as elaborated views of this basic design structure.
- In the following we describe our work in Darwin towards realising this approach and in particular relating analysis models to structural description.

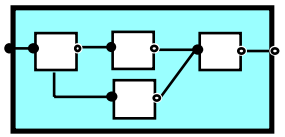
architecture description language (ADL) - Darwin

- **Darwin** describes ***structure***.
- **Darwin** architecture specification ***independent*** of component behaviour and component interaction.
- ***Framework*** for describing component behaviour, resource requirement, interaction type etc.
- **Darwin** used for ***specification***, ***construction*** and ***management***.

separation of concerns

Separate:

- **Configuration**
  hierarchic structure of system from component instances & interconnections

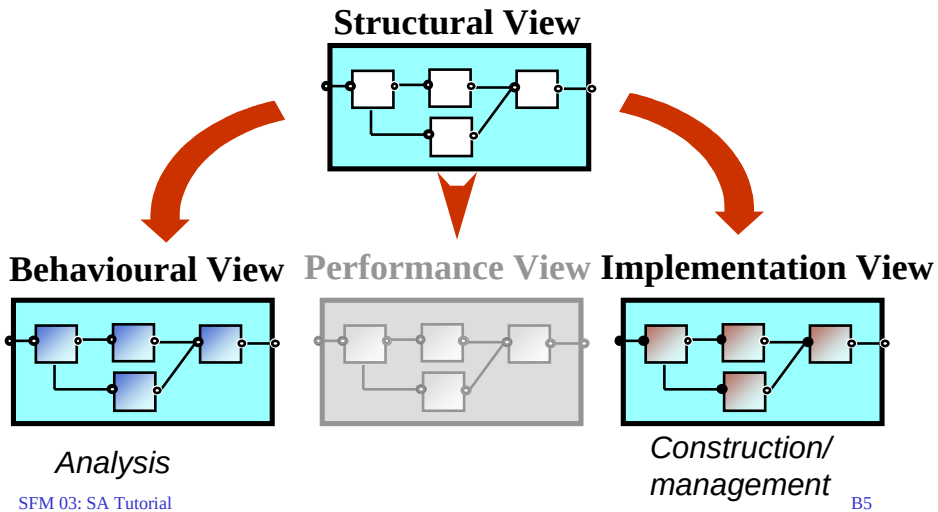


- **Communication**
  component interaction mechanisms

- **Computation**
  component behaviour

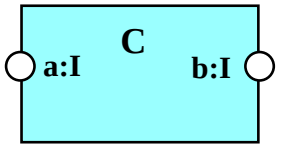
S1;S2

multi-view



structural view - components & interfaces

A component in Darwin can have one or more interfaces.




```
component C {
   portal a:I ;
          b:I ;
}
```

At this abstract level, an interface is simply a set of names:

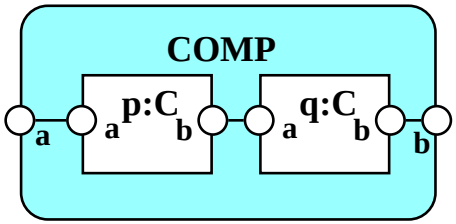
```
interface I {
        x ;
        y;
        z;
}
```

These will refer to actions in a specification or functions in an implementation.

structural view - composites & binding

Composite components are constructed from more primitive components using **inst** - instantiation & **bind** - binding.

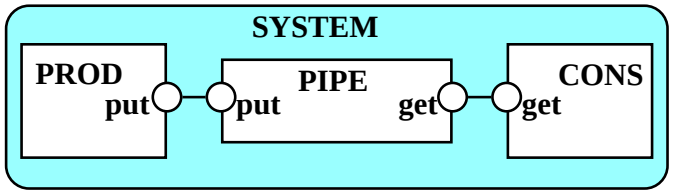
Portal types are inferred where they are not directly specified.




```
component COMP {
   portal a; b;
   inst  p:C;
         q:C;
   bind    p.a -- a;
           q.b -- b;
           p.b -- q.b;
}
```


structural view - connectors

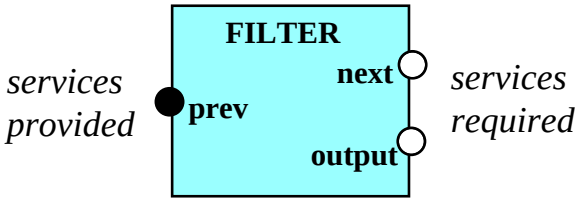
Darwin, in contrast to Wright & Unicon, does not have additional syntax to denote connectors. A connector is a type of component:




```
component PIPE {
   portal put;
          get;
}
```

service view - provide & require

The service view refines a **portal** into either a service **provide**d by a component or a service **require**d by a component.



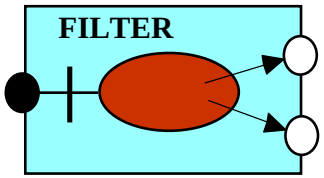

```
component FILTER {
   provide  prev : IntStream;
   require   next : IntStream;
             output : IntStream;
}
```


service view - towards implementation

In a distributed system, interfaces can be specified in Corba IDL and primitive components implemented as CORBA objects:

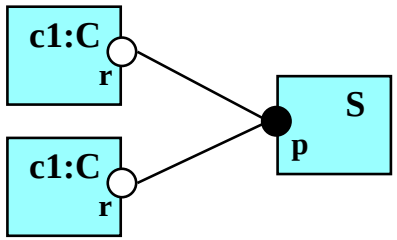
CORBA IDL

```
interface IntStream {
        void put(in long x);
}
```



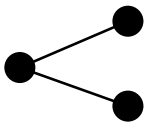
service view - binding patterns

many-to-one (e.g. client - server)

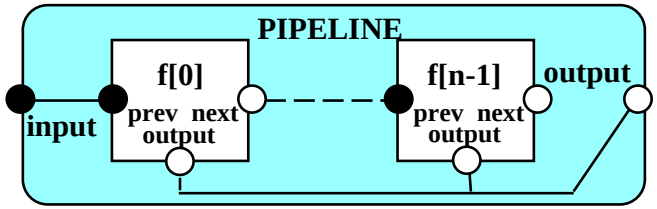



```
inst c1:C;
     c2:C;
     S;
bind
   c1.r -- S.p;
   c2.r -- S.p;
```

For replicated services only:



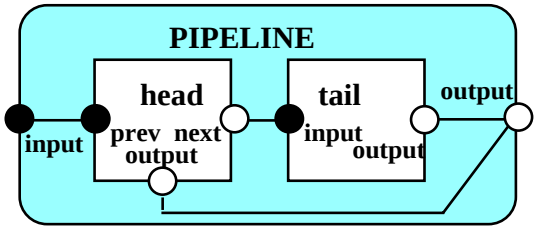
replicators (forall) and guards (when)



Dimensioning & Variants

```
component PIPELINE(int n) {
  provide input: IntStream;
  require output:IntStream;
  forall i = 0 to n-1 {
     inst f[i]:FILTER;
     bind f[i].output -- output;
     when i < n-1
        bind f[i].next -- f[i+1].prev;
  }
  bind input -- f[0].prev;
}
```

recursion

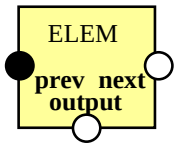



```
component PIPELINE(int n) {
  provide input;
  require output;
  inst head:FILTER;
  bind
      input -- head.prev;
      head.output -- output;
  when n>1 {
    inst tail:PIPELINE(n-1);
    bind
      head.next -- tail.input;
      tail.output -- output;
  }
}
```

Darwin structures are flattened at system build time such that composite components have no run time representation.

Consequently, defining structures using recursion has no impact on the resulting system runtime efficiency

generic components

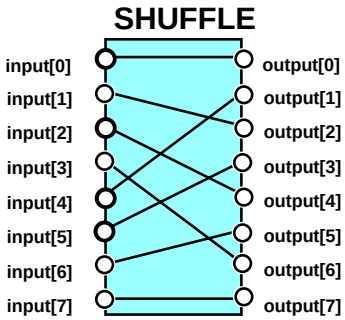


An analogy to a generic Darwin component is a printed circuit board which can be populated with chips which fit the sockets.

```
component ELEM {
   provide  prev;
   require   next; output;
}

component PIPELINE(int n, <ELEM>) {
  provide input;
  require output;
   forall i = 0 to n-1 {
      inst f[i]:<ELEM>;
      bind f[i].output -- output;
      when i < n-1
         bind f[i].next -- f[i+1].prev;
  }
  bind input -- f[0].prev;
}
```


binding components

Darwin components may contain only bindings. This is used to encapsulate complex interconnection patterns such as the perfect shuffle pattern shown below.




```
component SHUFFLE(int n) {
portal
        input[n];
        output[n];
forall k:0 to (n / 2)-1  bind
        input[k]  -- output[k*2];
        input[k+(n/2)] -- output[k*2 + 1];
}
```


Darwin - summary

Main Constructs

component - declares a primitive or composite type.

interface - declares an interface type

portal - declares an interface instance

provide - declares a service provided by a component.

require - declares a service required by a component.

inst - declares an instance of a component.

bind - declares a binding from a requirement to a provision

Darwin - summary

Additional Constructs

forall - replicates structure.

when - guards structure.

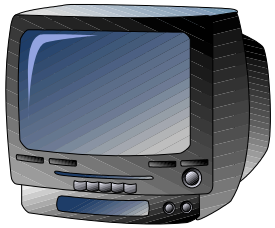
dyn - declares a set of dynamical created instances.

export - exports a service into a namespace.

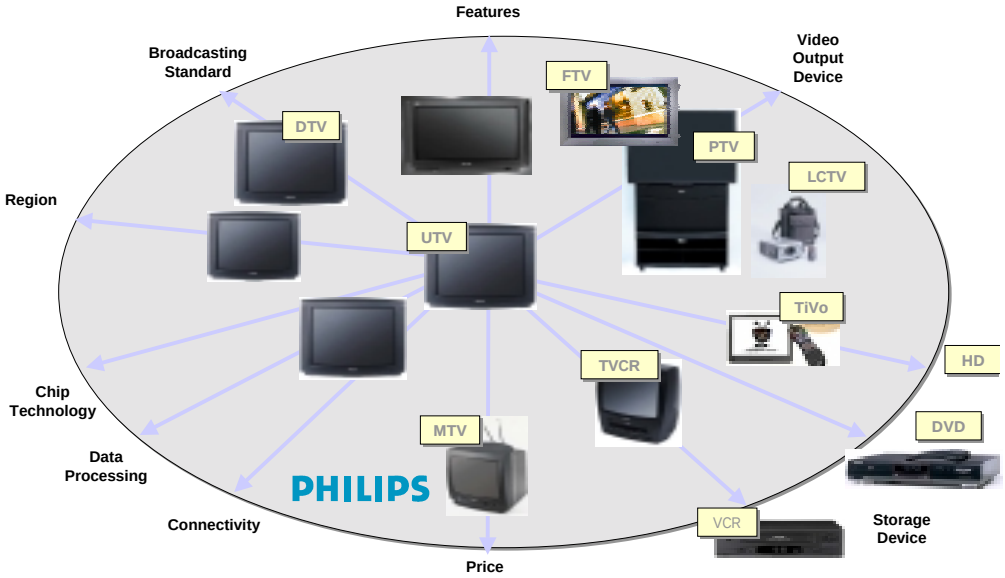
import - imports a service from a namespace.

Application - Televisions

Why is the Darwin ADL, which originated in distributed systems research, applicable to the construction of software for televisions?



Product Families



Role of an ADL...

- Uneconomic to design the software for each product from scratch.
- Develop a set of software components.
- Build the software for each product variant from an architectural description of that product.

Darwin applicability…

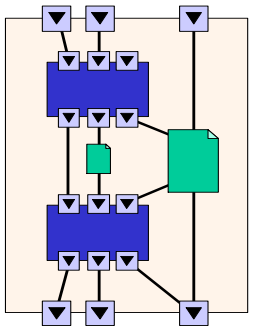
- Darwin enforces a strict separation between architecture and components.
- Variation supported by both different Darwin descriptions and parameterisation.
- Variants can be constructed at compile-time or later at system start-time.

Koala

In the ARES project
Rob van Ommering saw potential of Darwin in specifying television product architectures and developed Koala, based on Darwin, for Philips.

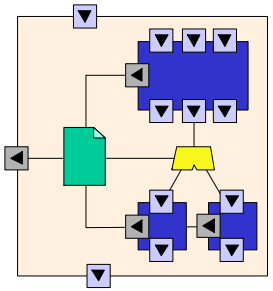
First large-scale industrial application of an ADL.

An industrial application of Darwin…



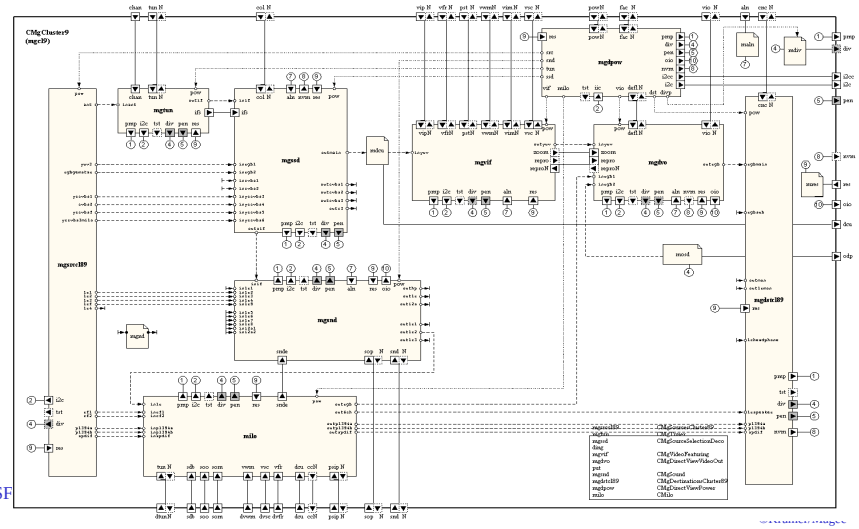
Interfaces are sets of C functions

Structural diversity
=
internal diversity
at next higher level



Resource constrained product families

Koala - example

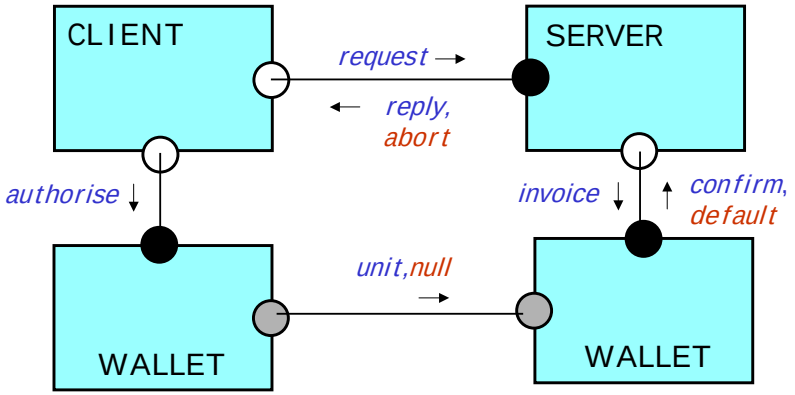


Darwin & Behaviour Models

- The basic approach is to use the Darwin structural description to generate FSP compositions using the static combinators - parallel composition ||, renaming / and hiding @.
- Behavioral descriptions using the dynamic combinators - action prefix -> and choice | are associated with primitive components.
- Interfaces are modelled by sets of actions.
- *Currently not all of Darwin is translatable into FSP.*

A simple *e*-commerce system…

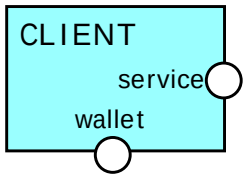
Client requests service from server which is paid for by a transfer from client's wallet to server's wallet.



Interfaces

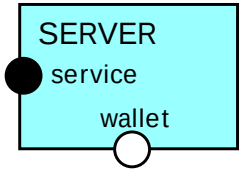
```
interface Wallet  {
    authorise;   // client authorisation to release funds
    invoice;     // server request to its wallet to get funds
    confirm;     // server wallet confirmation of fund transfer
    default;     // server wallet notification of failed transfer
    }
interface Service {
    request;     // request for service
    reply;       // successful provision of service
    abort;       // service not supplied
    }
interface Transfer {
    unit;       // transfer of one unit of funding
    null;       // no transfer of funds
    }
```


Client Component



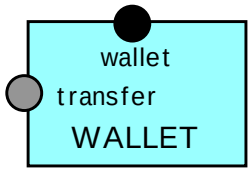

```
component CLIENT {
   require
     wallet:Wallet;
     service:Service;
 /%
 CLIENT =
   (wallet.authorise -> service.request ->
      (service.reply -> CLIENT
      |service.abort -> CLIENT
      )
    )+{wallet.Wallet,service.Service}.
 %/
 }
```

Server Component



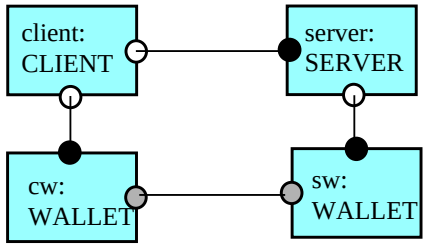

```
component SERVER {
  provide
    service:Service;
  require
    wallet:Wallet;
  /%
  SERVER
    = (service.request -> wallet.invoice
       -> (wallet.confirm -> service.reply ->SERVER
          |wallet.default -> service.abort ->SERVER
          )
     )+{wallet.Wallet,service.Service}.
  %/
  }
```


Wallet Component




```
component WALLET(int A) {
   provide wallet:Wallet;
   portal  transfer:Transfer;
/% WALLET(A=0) = WALLET[A],
   WALLET[a:0..Max]
       = (balance[a]->
          (wallet.authorise ->
           (when(a>0)  transfer.unit   -> WALLET[a-1]
           |when(a==0) transfer.null   -> WALLET[a]
           )
          |wallet.invoice ->
           (transfer.unit -> wallet.confirm -> WALLET[a+1]
           |transfer.null -> wallet.default -> WALLET[a]
           )
          )
         )+{wallet.Wallet,transfer.Transfer}.
 %/
 }
```


System Composition - Darwin




```
component SYS {
   inst
     client:CLIENT;
     server:SERVER;
     cw:WALLET(2);
     sw:WALLET(0);
   bind
     client.service -- server.service;
     client.wallet  -- cw.wallet;
     server.wallet  -- sw.wallet;
     cw.transfer    -- sw.transfer;
 }
```


System Composition - FSP

```
||SYS =
    (client:CLIENT
    || server:SERVER
    || cw:WALLET(2)
    || sw:WALLET(0)
    )
    /{client.service/server.service,
      client.wallet/cw.wallet,
      server.wallet/sw.wallet,
      cw.transfer/sw.transfer
     }.
```

Relabeling operator renames action name prefixes

Properties

Safety

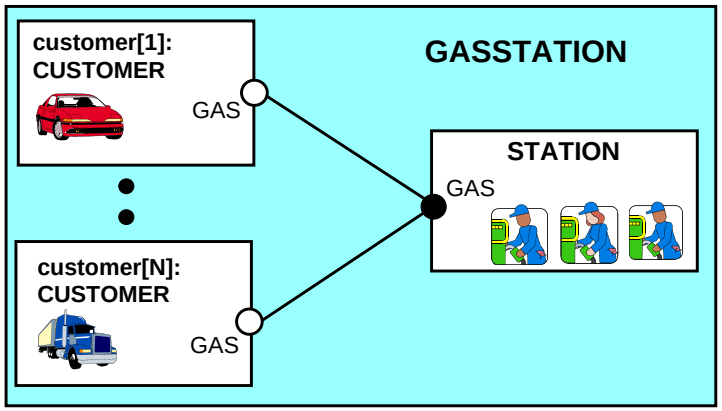
```
/* If a payment transfer occurs the service should be delivered
   otherwise if no payment, no service */
property HONEST
 = (cw.transfer.unit -> client.service.reply -> HONEST
   |cw.transfer.null -> client.service.abort -> HONEST
   ).
||CHECK = (SYS || HONEST).
```

Liveness

```
/* It should always be the case that the service either
   eventually replies or aborts */
LIVE_SERVICE = {service.{reply,abort}}
```


Gas Station Example

Client - Server Architecture in which STATION encapsulates a set of PUMP servers managed by a CASHIER.

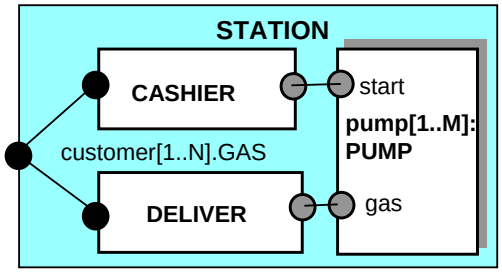


Gas Station - Darwin

```
int N = 3; // number of customers
int M = 2; // number of pumps

interface GAS {
  prepay; // prepay amount for gas
  gas;    // gas delivered
}

component GASSTATION {
  inst
    STATION;
  forall i = 1 to N {
    inst
      customer[i]:CUSTOMER;
    bind
      customer[i].SERVICE --STATION.customer[i];
  }
}
```


STATION component

```
component STATION {
  provide
    customer[1..N]:GAS;
  inst
    CASHIER;
    DELIVER;

  forall i = 1 to N {
    bind
      customer[i].prepay -- CASHIER.customer[i].prepay;
      customer[i].gas    -- DELIVER.customer[i].gas;   }
  forall i = 1 to M {
    inst pump[i]:PUMP;
    bind
      pump[i].start -- CASHIER.start[i];
      pump[i].gas   -- DELIVER.gas[i];   }
}
```



CUSTOMER & PUMP components

```
/%
range C = 1..N  //customer range
range P = 1..M  //pump range
range A = 1..2  //amount of money or Gas
%/

component CUSTOMER {
  require GAS;
/%
CUSTOMER = (prepay[a:A]->gas[x:A]->
              if (x==a) then CUSTOMER else ERROR). %/
}

component PUMP {
  portal start; gas;
/%
PUMP = (start[c:C][x:A] -> gas[c][x] -> PUMP). %/
}
```


CASHIER & DELIVER components

```
component CASHIER {
  provide customer[1..N]:GAS;
  portal  start[1..N];
/%
CASHIER
  = (customer[c:C].prepay[x:A] -> start[P][c][x] -> CASHIER).
%/
}

component DELIVER {
  provide customer[1..N]:GAS;
  require gas[1..M];
/%
DELIVER=
  (gas[P][c:C][x:A] -> customer[C].gas[x] -> DELIVER).
 %/
}
```


Find the bug!

```
Composition:
GASSTATION = STATION || customer.1:CUSTOMER ||
customer.2:CUSTOMER || customer.3:CUSTOMER
State Space:
 4459 * 3 * 3 * 3 = 2 ** 19
Analysing...
Depth 5 -- States: 81 Transitions: 242 Memory used: 3632K
Trace to property violation in customer.2:CUSTOMER:
      customer.1.prepay.1
      tau
      tau
      customer.2.prepay.2
      customer.2.gas.1
Analysed in: 0ms
```

DEMO...

Darwin+Models Further Reading

[1] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. "Specifying Distributed Software Architectures", in *Proc. of the 5th European Software Engineering Conference (ESEC'95)*. September 1995, Sitges, Spain, Lecture Notes in Computer Science 989, pp. 137-153. W. Schäfer and P. Botella, Eds.

[2] Magee, J. and Kramer, J. "Dynamic Structure in Software Architectures", in *Proc. of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 4)*. October 1996, San Francisco, California, USA. ACM Press, Software Engineering Notes 21, pp. 3-14. D. Garlan, Ed.

[3] Magee, J., Kramer, J., and Giannakopoulou, D. "Analysing the Behaviour of Distributed Software Architectures: a Case Study", in *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*. October 1997, Tunis, Tunisia, pp. 240-245.

[4] Kramer, J. and Magee, J., *Analysing Dynamic Change in Distributed Software Architectures*. IEE Proceedings - Software, Vol. **145**(5): pp. 146-154.

[5] Magee, J., Kramer, J., and Giannakopoulou, D. "Behaviour Analysis of Software Architectures", in *Proc. of the 1st Working IFIP Conference on Software Architecture (WICSA1)*. 22-24 February 1999, San Antonio, TX, USA.

[6] Van Ommering R., van der Linden F., Kramer J., and Magee J., "The Koala Component Model for Consumer Electronics Software ", *IEEE Computer* 33 (3), March 2000, 78-85.

[7] Kramer J. and Magee J., "Distributed Software Architectures: Tutorial", *20th IEEE Int. Conf. on Software Engineering (ICSE-20)*, Kyoto, April 1998, Volume II, 280-281.

Part C :
Animation & Analysis

- Graphic Animation
- Fluent LTL Model Checking

Graphic Animation - Motivation

The products of analysis are essentially action traces describing desirable or undesirable behaviours that the model has.

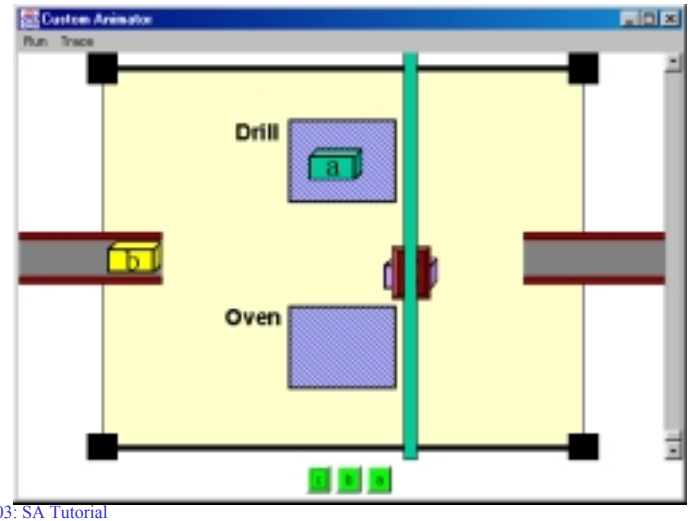
The purpose of graphic animation is to provide visualizations of these behaviours.

These visualizations can be in the context of the architecture or in the context of the problem domain.

Graphic Animation - Outline

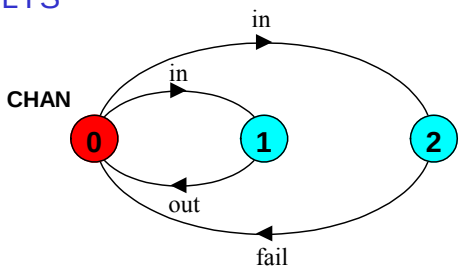
- Motivating example
- Timed Automata Framework
- Animation Composition
- SceneBeans – animation engine
- Directions

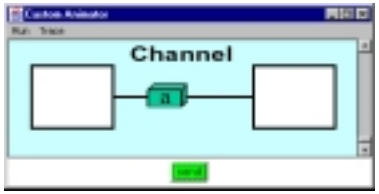
Flexible Production Cell



A simpler example- CHAN

LTS



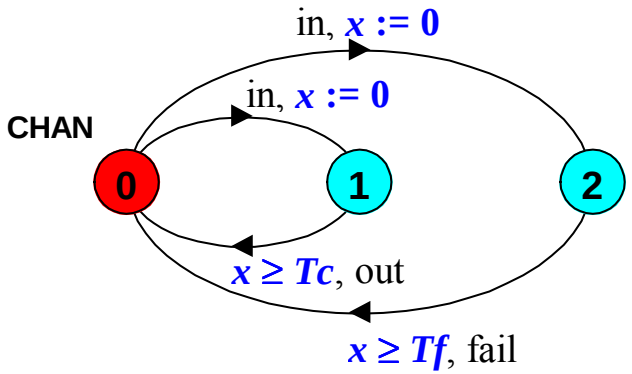


FSP

```
CHAN = (in -> out  -> CHAN
       |in -> fail -> CHAN
       ).
```


Timed Automata

Abstract animation activities by local clocks that measure the passage of time.



local clock variable *x*

Time passes in a state.

Animation Activities

channel

commands:

channel.begin	-- corresponds to	$x := 0$
explode		

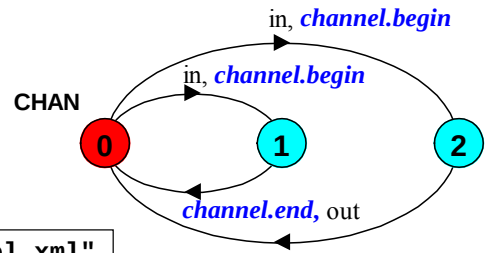
conditions:

channel.end	-- corresponds to	$x \geq Tc$
channel.fail	-- corresponds to	$x \geq Tf$

Start of an activity

Signal as the activity progresses or ends

Annotating LTS with animation



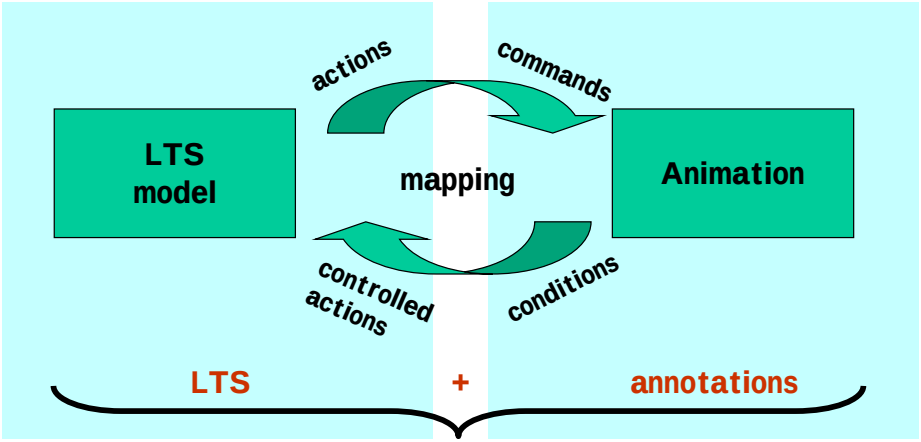
Mapping Relation

```
animation FAILCHAN = "channel.xml"
   actions {
      in   / channel.begin,
      fail / explode
   }
   controls {
      out  / channel.end,
      fail / channel.fail
   }
```

label/command (immediate actions)

label/condition (controlled actions)

Model-Animation Structure



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.

Progress properties

Useful approximation of the annotation is:

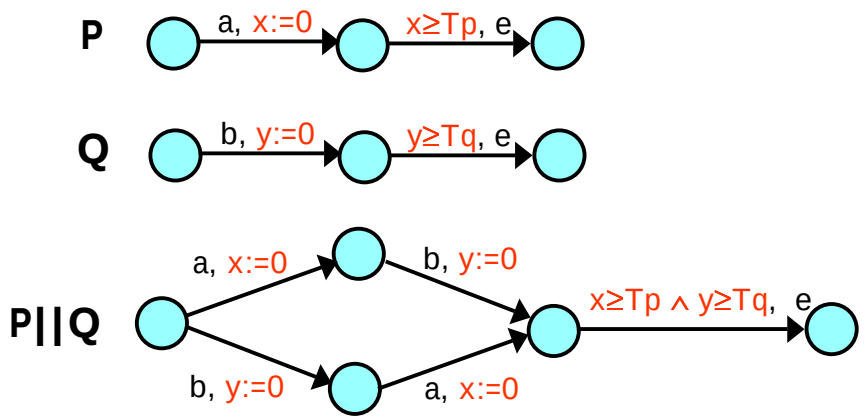
P>>*Controlled* -- make actions in Controlled low priority

Check

progress NOZENO = { *Controlled* }

asserts animation is free of Zeno executions.

Composition - Timed Automata



Animations can be composed in the same way.

Animation Composition

An animation is defined by;
- the set of commands ***C***,
- the set of conditions ***B***
- the relation ***Actions*** -- maps LTS actions to commands
- the relation ***Controls*** -- maps LTS actions to conditions

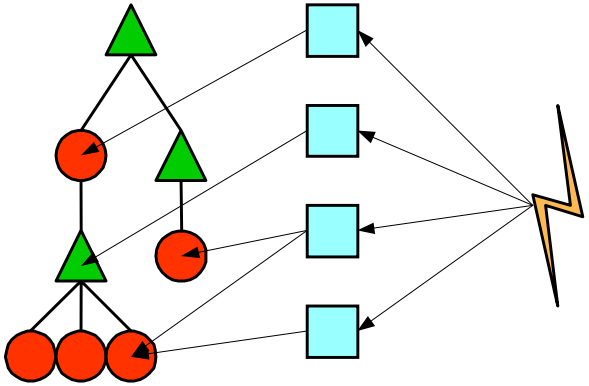
Animation Composition

animation $M_1 = \langle C_1, B_1, Actions_1, Controls_1 \rangle$
animation $M_2 = \langle C_2, B_2, Actions_2, Controls_2 \rangle$

animation $\boldsymbol{M_1 \| M_1} = \langle C_1 \cup C_2, B_1 \cup B_2, Actions_1 \cup Actions_2, Controls_1 \cup Controls_2 \rangle$

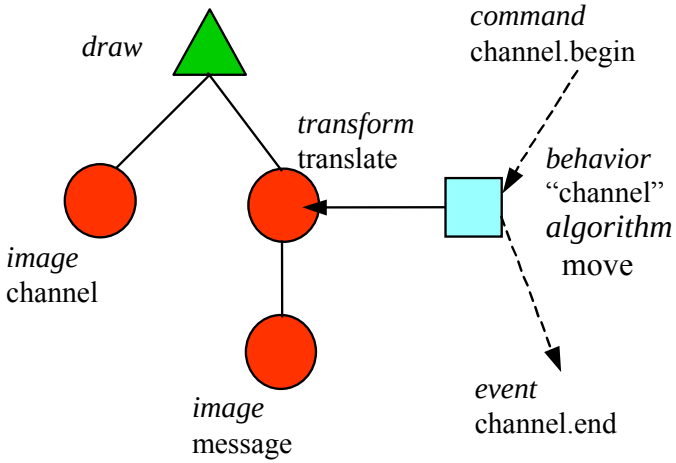
Example ..

SceneBeans



Scene Graph Behaviours Animation Thread

Example Scene Graph



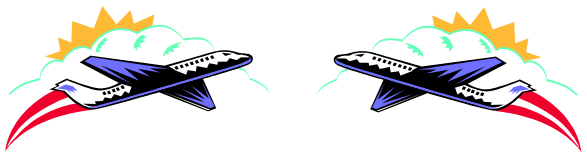
XML

```
<?xml version="1.0"?>
<!DOCTYPE animation SYSTEM "scenebeans.dtd">
<animation width="400" height="136">
<behavior id="channel" algorithm="move"
        event="channel.end">
   <param name="from" value="71"/>
   <param name="to"   value="323"/>
   <param name="duration" value="2"/>
</behaviour>
<command name="channel.begin">
  <announce event = "~channel.end"/>
  <start behaviour= "channel"/>
</command>
<event object="channel" event="channel.end">
  <announce event="channel.end"/>
</event>
<draw>
  <transform type="translate">
    <param name="y" value="64"/>
    <animate param="x" behavior="channel"/>
    <image src="image/message.gif"/>
  </transform>
  <image src="image/channel.gif"/>
</draw>
</animation>
```

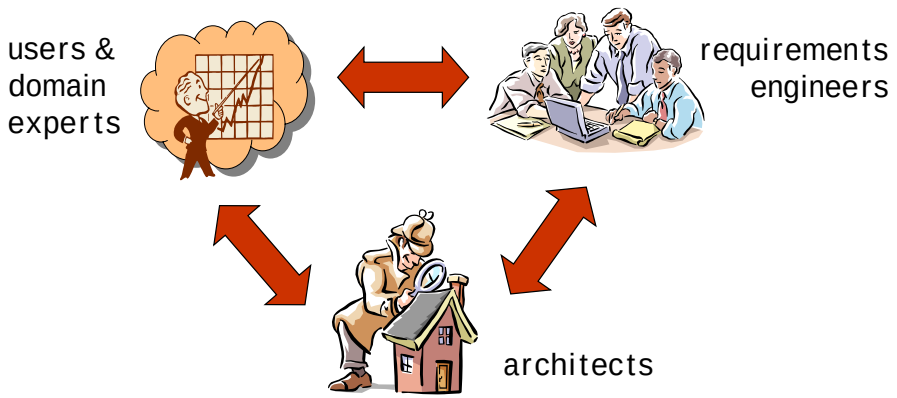

Animation Application

Air Traffic Control

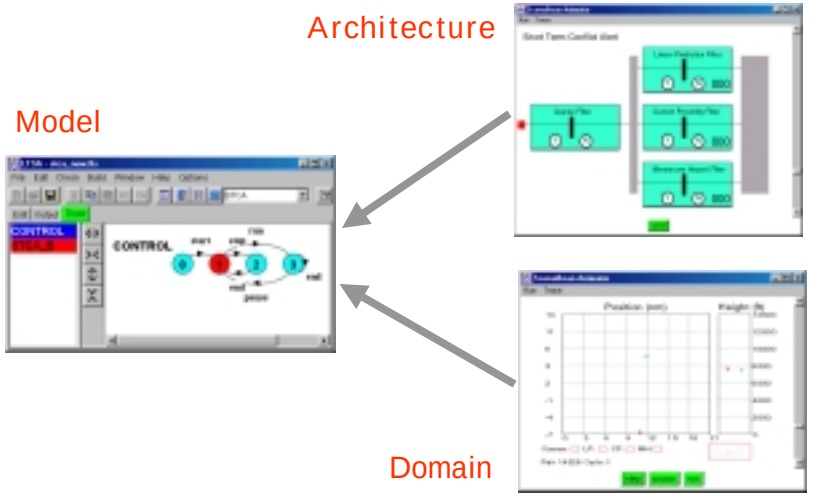
Short Term Conflict Alert



Animation..



Animations views...



Related Work

- Verification / Modelling Tools
  - **StateMate** – Widget Set
  - **SCR** – instrument panel animation
  - **SPIN, Concurrency Factory, UPPAAL** – animation w.r.t. model source
- Program Animation
  - **Tango/XTango** – smooth animation of sequential programs
  - **Pavane** – data parallel program animation via state/visual mapping

Graphic Animation - Conclusion

- Approach supports incremental and compositional development of animations.
- Clear separation between model and animation.
- Sound underlying semantics
- Requires animation design tool
  – XML not people friendly

*If **models** aid requirements capture, analysis and understanding, then so should model **animation**!*

Architecture Analysis

- Analyze complex systems at architectural level
  - structure in terms of components, connectors, bindings
  - behavior in terms of events / actions: messages sent or received, service invocations initiated or accepted by components
- Compositional Reachability Analysis based on architecture
  - allows for incremental minimization for state space reduction
- Properties expressed in terms of events
  - "each client request is followed by a reply by the server"

The Problem

Support the expression and analysis of temporal logic properties for an architecture where behavior is described in terms of events

Implementation in LTSA

- Component behavior described in the FSP language
- Semantics is labeled transition systems (LTSs)
  - an LTS M has a communicating alphabet αM
  - internal actions represented by action " τ "
- LTSs assembled with parallel composition operator "||"
  - synchronizes shared actions, interleaves remaining actions
- Support for safety and progress properties
- Wish to provide support for LTL
  - supported by specification patterns (Dwyer, Avrunin, Corbett)
  - widely used in practice, e.g. tools like SPIN

Linear Temporal Logic (LTL)

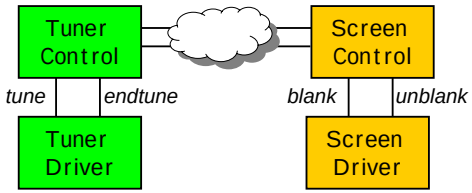
- LTL formulas built from:
  - atomic propositions in P and standard Boolean operators
  - temporal operators **X** (next time), **U** (strong until), **W** (weak until), **F** (eventually) and **G** (always)
- Interpreted on infinite words $w = \langle x_0 x_1 x_2 \ldots \rangle$ over 2^P
  - x_i is the set of atomic propositions that hold at time instant i
- An infinite execution of an LTS is an infinite sequence of actions that it can perform starting at its initial state

What are meaningful atomic propositions for an LTS?

Action LTL (ALTL)

- [Giannakopoulou 99] ; [Leuschel, Massart & Currie 01]
- P is the universal set of actions
  - for example, $\varphi \equiv \mathbf{G}\ (request \Rightarrow \mathbf{F}\ reply)$
- ALTL formulas interpreted on infinite words of actions (a single action holds at any time)
  - $\langle\ request, reply, \ldots, request^{\omega}\ \rangle$ – violates property φ
  - $\langle\ (request, reply)^{\omega}\ \rangle$ – satisfies property φ
- An LTS satisfies a formula φ if all its infinite executions satisfy φ

Example



to avoid artifacts on screen when changing channels, if the tuner is tuning, then the screen must be blanked

- if screen is initially blanked and the tuner is not tuning
  - NOARTIFACTS $\equiv \mathbf{G}\ (\ (unblank \Rightarrow (\neg tune\ \mathbf{W}\ blank))\ \wedge\ (tune \Rightarrow (\neg unblank\ \mathbf{W}\ endtune))\ )$
- if screen is initially not blanked
  - $(\neg tune\ \mathbf{W}\ blank)\ \wedge\ \mathbf{G}\ (\ (unblank \Rightarrow (\neg tune\ \mathbf{W}\ blank))\ \wedge\ (tune \Rightarrow (\neg unblank\ \mathbf{W}\ endtune))$

Limitation

- properties refer to state predicates
  - "screen is blank", "tuner is tuning"
- actions define intervals where predicates are true or false
  - "screen is blank" is true after action "blank" and before "unblank"
  - often multiple actions may initiate the same predicate
- formulas express relationships between these intervals
- complexity of expression grows significantly as more predicates are introduced
  - e.g. NOARTIFACTS has to hold only in a specific mode of the TV
- formula is different depending on initial state of predicate(s)
- introduce **fluents!**

Fluents

[Sandewall 94]; [Kowalski, Sergot 86]; [Miller, Shanahan 99]

Fluents (time-varying properties of the world) 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. Similarly, a fluent is false at a particular time-point if it has been previously terminated and not initiated in the meantime

Fluent LTL (FLTL)

- Set of atomic propositions is set of fluents Φ
- We define fluents as follows: $Fl \equiv \langle I_{Fl}, T_{Fl} \rangle$
  - I_{Fl}, T_{Fl} are sets of initiating and terminating actions accordingly, such that $I_{Fl} \cap T_{Fl} = \varnothing$
- A fluent *Fl* may initially be true or false at time zero as denoted by the attribute *InitiallyFl*
- For LTS *M,* an action *a* defines implicitly:
  - *Fluent(a)* $\equiv \langle \{a\}, \alpha M - \{a\} \rangle$ Initially_a = false

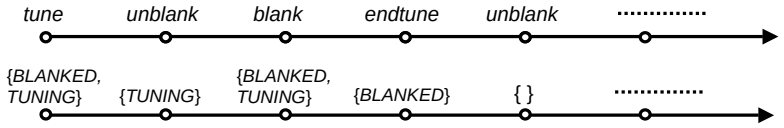
TUNING $\equiv \langle \{tune\}, \{endtune\} \rangle$ $Initially_{TUNING}$ = false

BLANKED $\equiv \langle \{blank\}, \{unblank\} \rangle$ $Initially_{BLANKED}$ = true

NOARTIFACTS $\equiv$ **G**(*TUNING* $\Rightarrow$ *BLANKED*)

FLTL

- Interpreted on infinite words over 2^{Φ}
- Infinite executions of LTSs define infinite words over 2^{Φ}
  - actions have immediate effect on the values of fluents
  - a fluent holds at a time instant iff (1) it holds initially or some initiating action has occurred and (2) no terminating action has occurred
- From TV example:



TUNING $\equiv \langle \{tune\}, \{endtune\} \rangle$ $Initially_{TUNING}$ = false

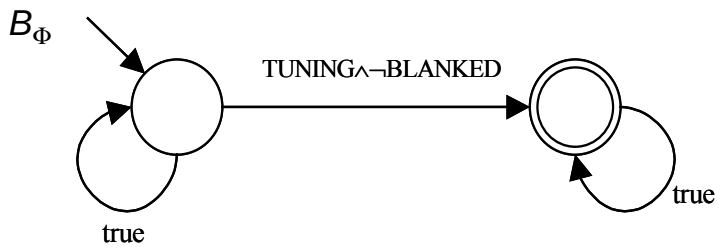
BLANKED $\equiv \langle \{blank\}, \{unblank\} \rangle$ $Initially_{BLANKED}$ = true

Model Checking FLTL

- Model check LTL formula φ on a system *M*:
  1. construct a Büchi automaton *B* for ¬φ
  2. check emptiness of *synchronous product* $M \times B$
- Executions of LTSs are defined over actions; FLTL formulas are defined over fluents
  - [De Nicola, Vaandrager 95] introduce DTSs
  - our approach: fluents define state predicates implicitly and are model checked using standard operators of LTSs
- Model check FLTL formula φ on system *M*:
  1. construct a Büchi automaton *B* for ¬φ
  2. augment *M* with values of fluents that hold at each state
  3. check emptiness of synchronous product

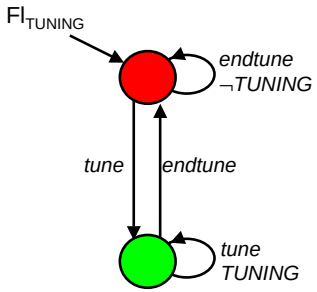
Büchi construction

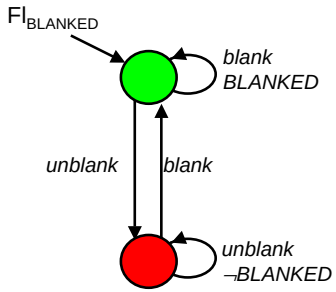
- [Giannakopoulou, Lerda 02]
- construct a complete automaton



¬**G** (*TUNING* $\Rightarrow$ *BLANKED*)

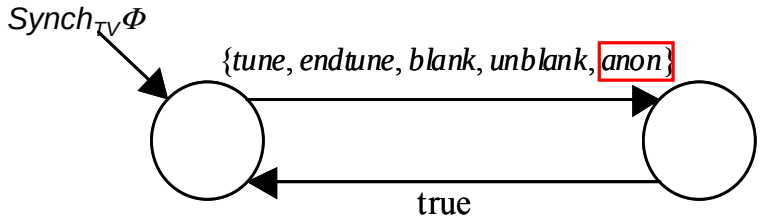
Adding Fluent Labels





If *TV* is the LTS for our example, then:
TV_Augmented = (*TV* || FL_{TUNING} || $FL_{BLANKED}$)

Synchronous Product



CHECK_TV = (*TV* || FL_{TUNING} || $FL_{BLANKED}$ || $Synch_{TV}\Phi$ || $B\Phi$)

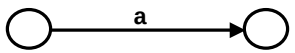
Tester Automaton

CHECK_TV = (*TV* || FL_{TUNING} || $FL_{BLANKED}$ || $Synch_{TV}\Phi$ || $B\Phi$)

- second part of composition is *fairly* independent
- create "tester" automaton
  - combines fluents, synchronizer and Büchi automaton
  - can be optimized and reused
- optimization steps for
  - synchronous product
  - safety properties
  - partial order reduction

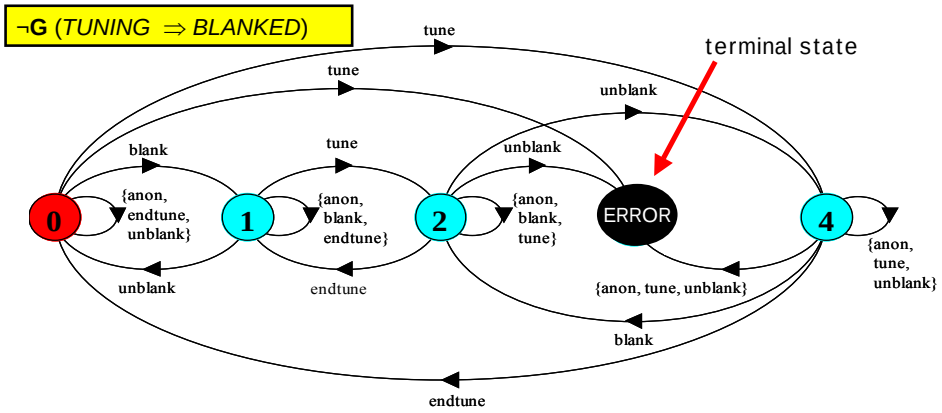
Synchronous product

- Our construction computes intermediate states where Büchi automaton has not yet observed the system state
  - hide all labels in 2^{Φ}
  - merge consecutive transitions consisting of a non-τ transition followed by a τ transition



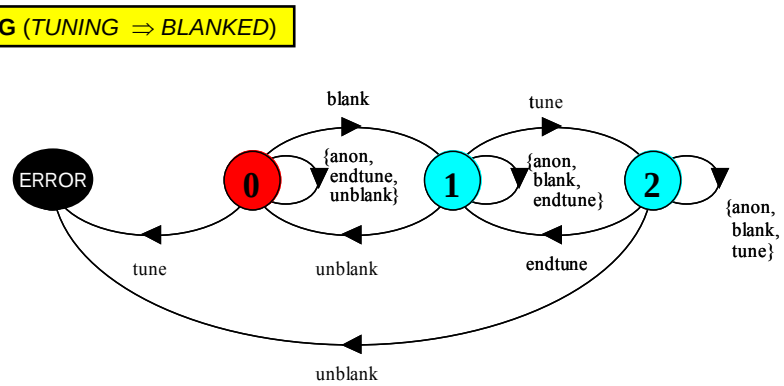
  - minimize with respect to strong bisimulation

Safety Properties



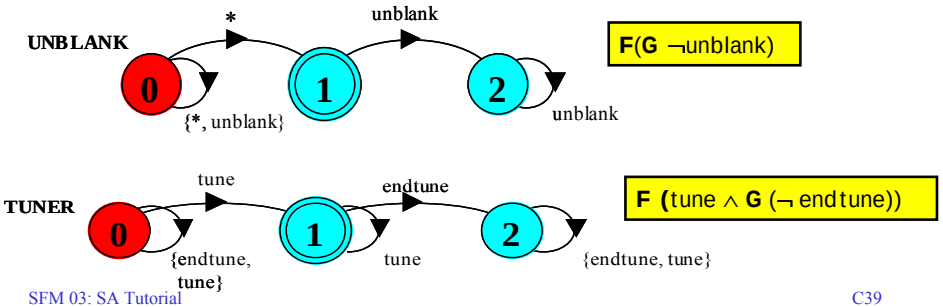
- all accepting states terminal ⇒ a pure safety property
- safety properties can be made deterministic and minimal

Safety properties



Partial Order Reduction

- based on notion of "independence" between transitions
  - when tester observes all events in the system, all transitions become dependent on each other
- turn all actions that do not appear in the definition of fluents into "*"
  - introduce special rules for parallel composition to deal with *
- remove * loops in certain cases



LTSA

```
fluent BLANKED =  <blank,unblank> initially True
fluent TUNING  = <tune,endtune>
assert NOARTIFACTS = [](TUNING -> BLANKED)
```

```
fluent CRITICAL[i:1..2] = <p[i].enter, p[i].exit>
assert MUTEX = []!(CRITICAL[1] && CRITICAL[2])
```

```
Trace to property violation in MUTEX:
  p[1].mutex.down
  p[1].enter CRITICAL[1]
  p[2].mutex.down CRITICAL[1]
  p[2].enter CRITICAL[1] && CRITICAL[2]
Analysed in: 20ms
```

Fluent Model Checking - Conclusion

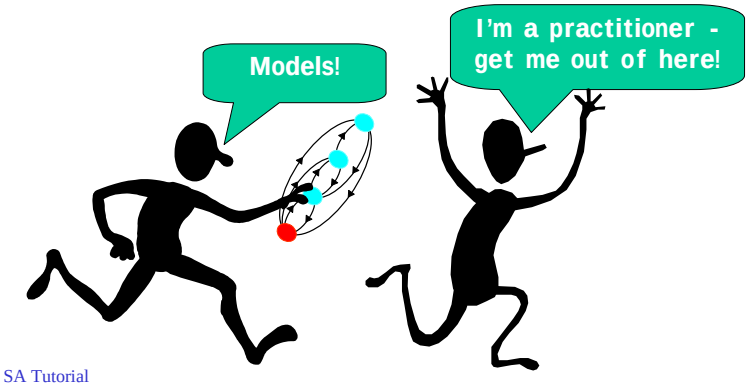
- Fluents
  - elegantly introduce state in event-based systems without changing the original model
  - allow for concise expression of temporal properties
  - [Paun, Chechik 99] added events to state-based systems
- Model checking FLTL properties
  - construction and optimization of tester automata
  - weaken the dependence between tester and system actions to permit partial order reduction
  - currently investigate conditions that could allow further reduction
- Have proven very useful in our case studies
- Implemented in LTSA tool - available for download

Further Reading

[1] Magee, J., Pryce, N., Giannakopoulou, D., and Kramer, J. "Graphical Animation of Behavior Models", in *Proc. of the ICSE*. June 2000, Limerick. IEEE.

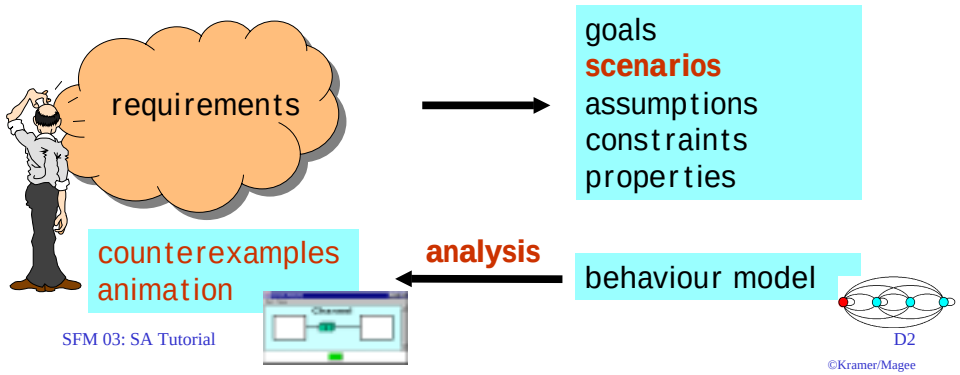
[2] Giannakopoulou, D. and Magee, J. "Fluent Model Checking for Event-based Systems", in *Proc. of the 9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Sept. 2003, Helsinki. ACM.

Part D :
Models from Scenarios



Motivation – we believe that ....

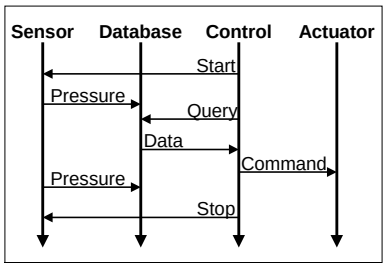
- Model construction should be part of requirements process
- early identification of an outline component architecture & behaviour model helps to inform the requirements
- analysis by model checking and animation of model behaviour and misbehaviour (property violations).



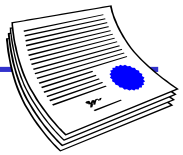
Scenarios

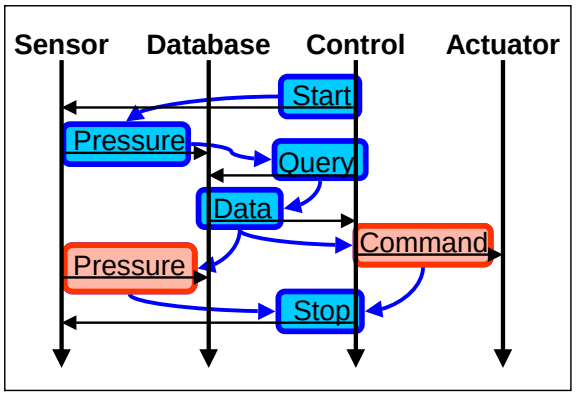
- Widely used for requirements elicitation and documenta
- Partial stories about components and how they interact
- Used in sets to describe system behaviour.
- BUT are generally used in an informal way with no precise semantics

=> ***Need scenarios and models with complementary semantics.***



Basic MSC - Message Sequence Chart

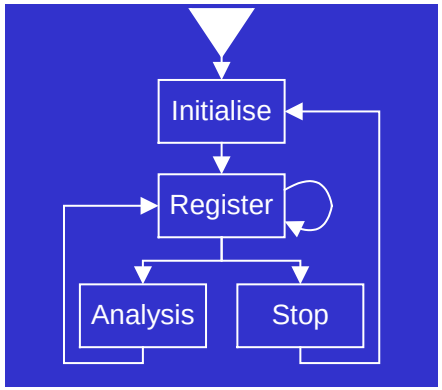


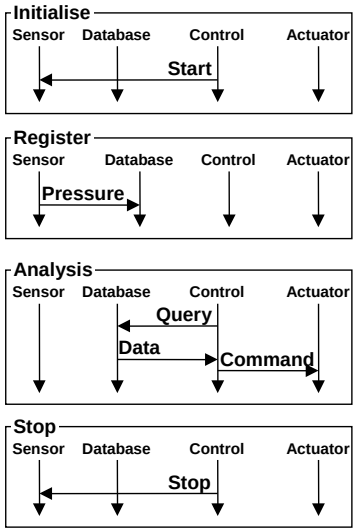


- Widely accepted notation.
- Standard: ITU & UML Sequence Diagrams.
- **Components, messages** and time.
- Synchronous communication
- Partial order semantics.

Start, Pressure, Query, Data, **Command, Pressure**, Stop.
Start, Pressure, Query, Data, **Pressure, Command**, Stop.

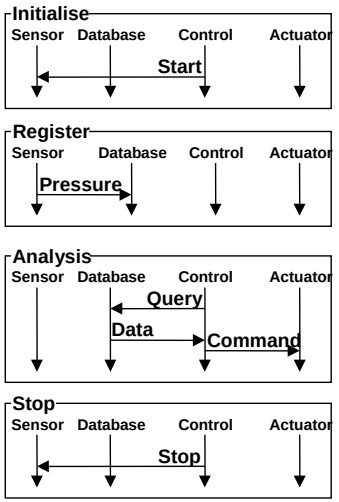
High level MSC

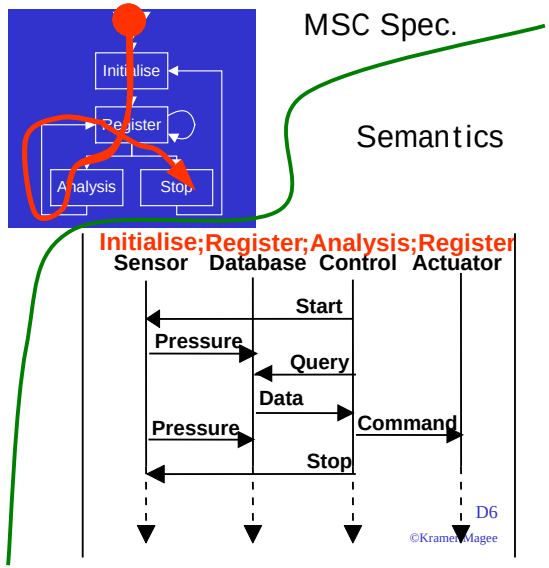




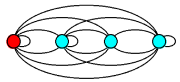
- Nodes are bMSCs or hMSCs.
- Scenario reuse and scalability.
- ITU Standard/Not UML.

High level MSC semantics



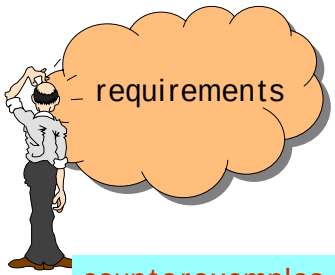


Model definition?



- if model construction is part of requirements process => how can we arrive at an outline behaviour model?

Some form of Synthesis?



Scenarios (MSC)

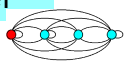
synthesis

architecture behaviour model

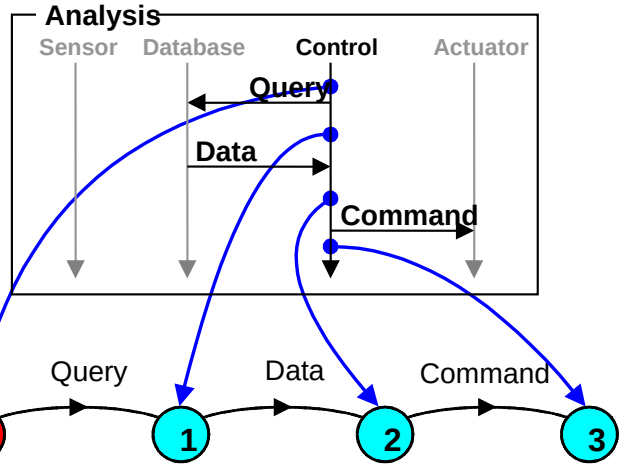
analysis

counterexamples animation

Composition of component behaviours

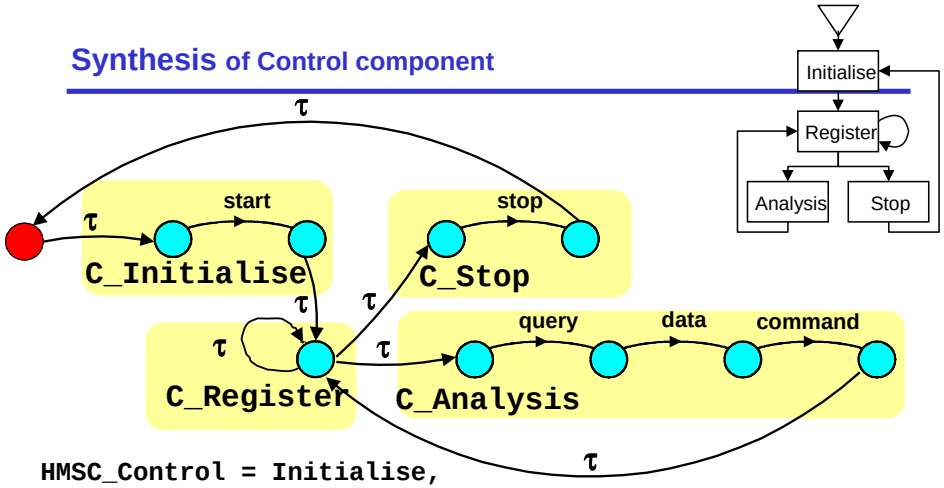


Synthesis of Control component



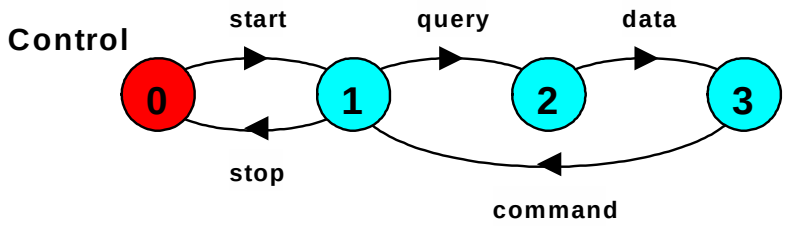

```
C_Analysis = (query->data->command->End)
```

Synthesis of Control component



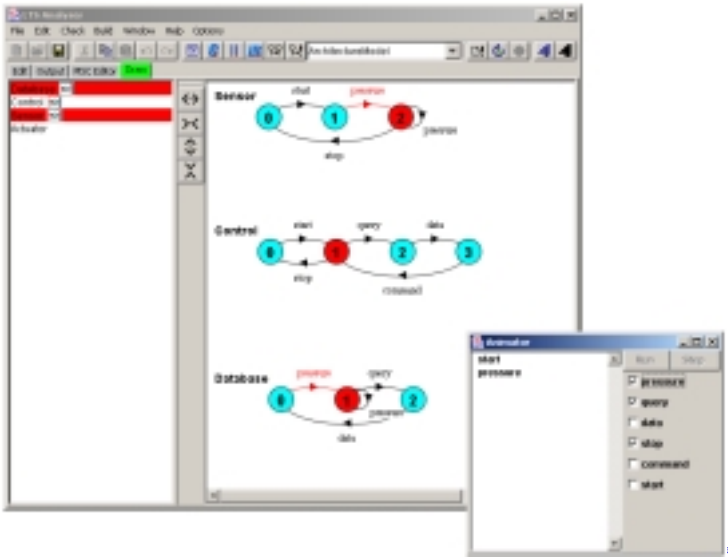

```
HMSC_Control = Initialise,
Initialise = C_Initialise;Next_Initialise,
Register = C_Register;Next_Register,
...
Next_Initialise = (t->Register),
Next_Register = (t->Register | t->Analysis | t->Stop).
```


Synthesis of Control component



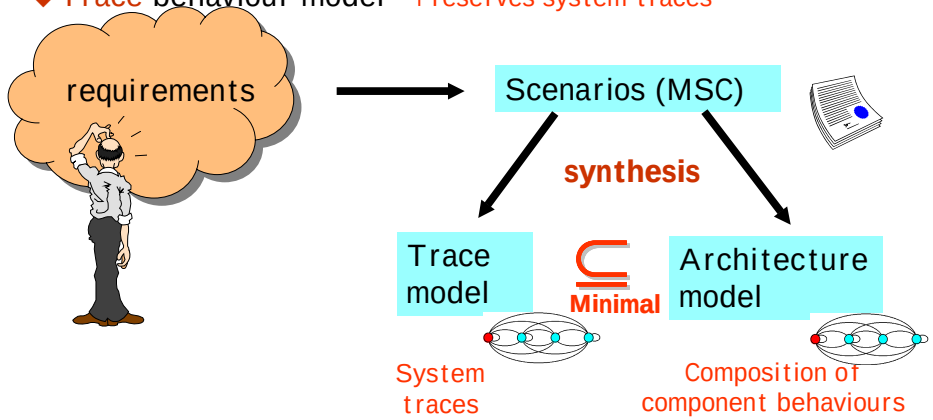

```
deterministic ||Control = HMSC_Control\{t}.
```


architecture behaviour model - animation



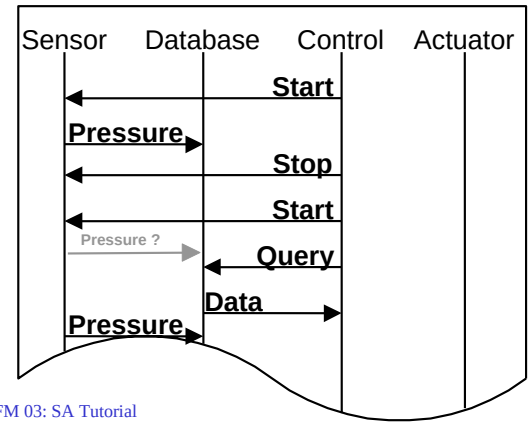
Synthesis of models

- Architectural behaviour model - Composition of components
- Trace behaviour model – Preserves system traces



Additional traces? Implied Scenarios

- Architecture model introduces additional system behavior through lack of sufficient local information.



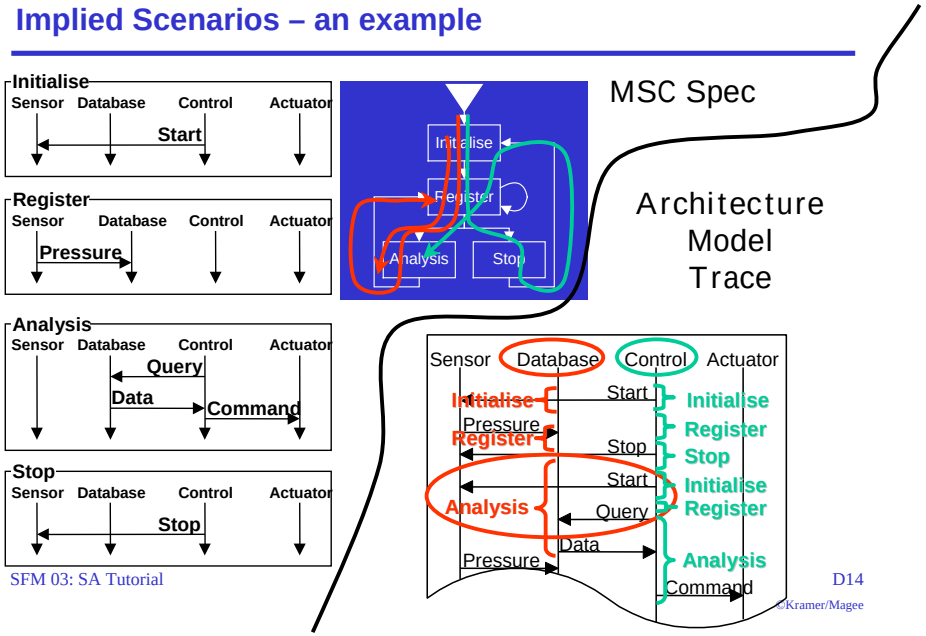
Control cannot observe when Database has registered data

=> Database should enable/disable queries.

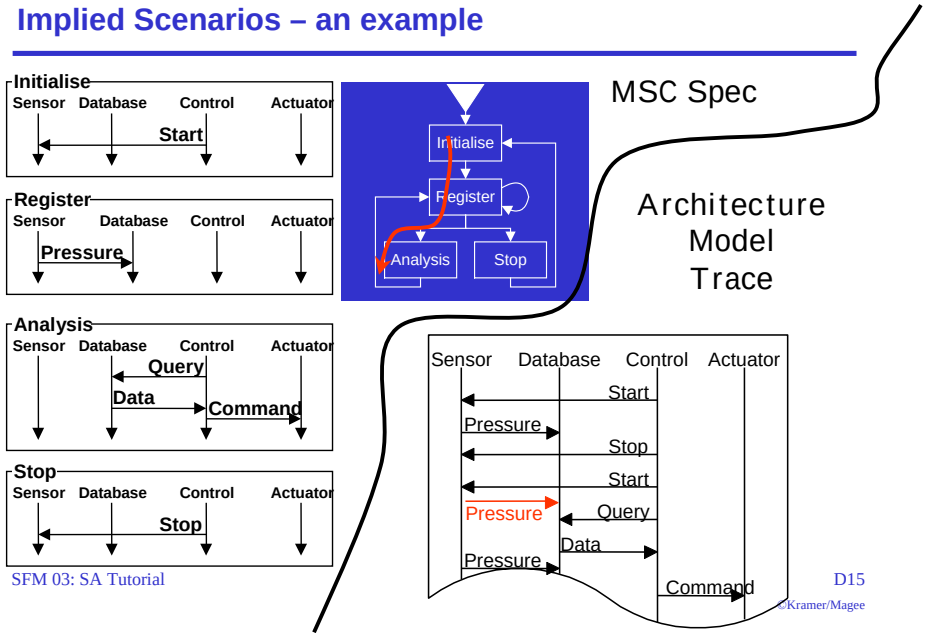
But Database cannot observe when sensor has been started and stopped.

Acceptable ?

Implied Scenarios – an example



Implied Scenarios – an example



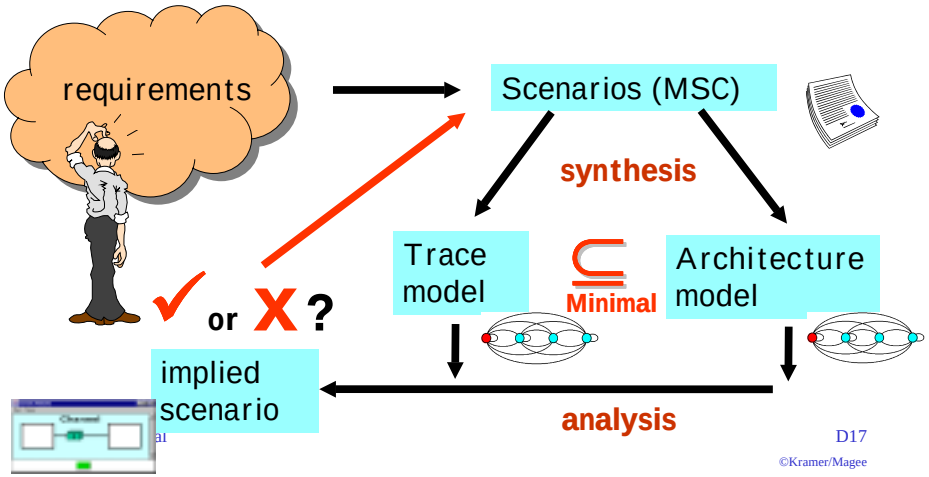
Implied Scenarios...

- Result from a mismatch between specified behaviour and architecture.
- Which one is wrong?
  - Missing scenario
  - Incorrect or too abstract architecture
- Implied scenarios are “gaps” a the MSC specification!

Implied scenarios should be ***detected*** and ***validated***

Synthesis and Elaboration

- Synthesis of the models, analysis by model checking, elaboration aided by implied scenarios.

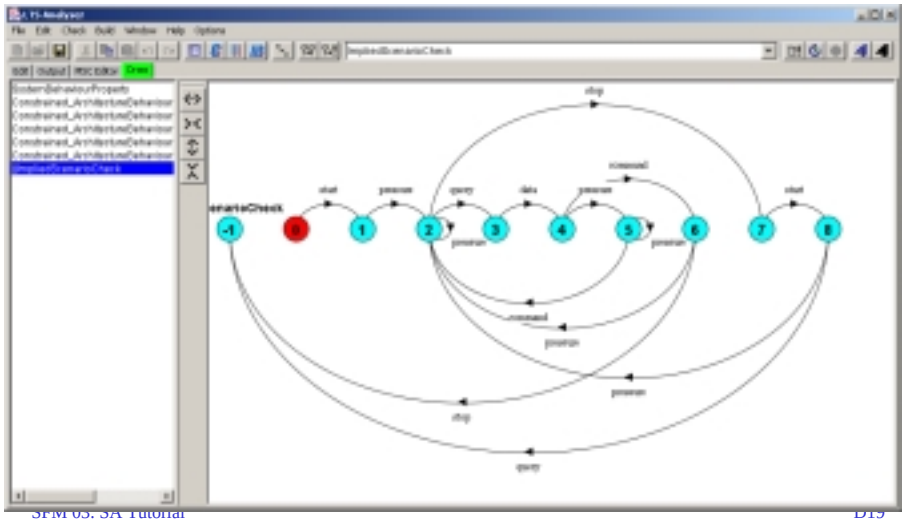


Implied Scenario Detection

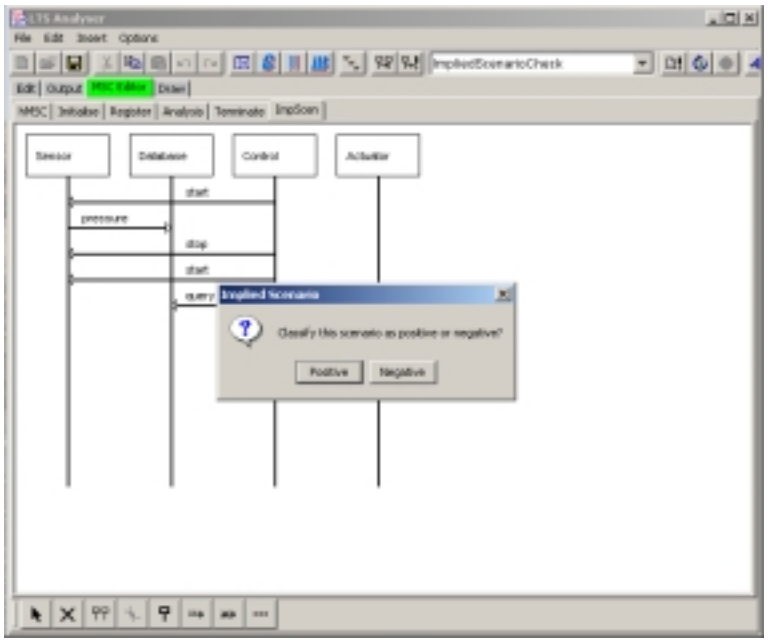
- Build model *Trace Model* T s.t "tr(T) =L(Spec)"
  - Ignore component structure
  - Non-trivial
    - Weak bMSC sequential composition
    - Possibly non-regular MSC language
- Model check "tr(A) ⊆ tr(T)"
  - Declare T as safety property
  - Check for reachability of error state in (T||A)
- Counter-examples are implied scenarios

c.f. [FSE'02]

Implied scenario detection



Implied scenario query



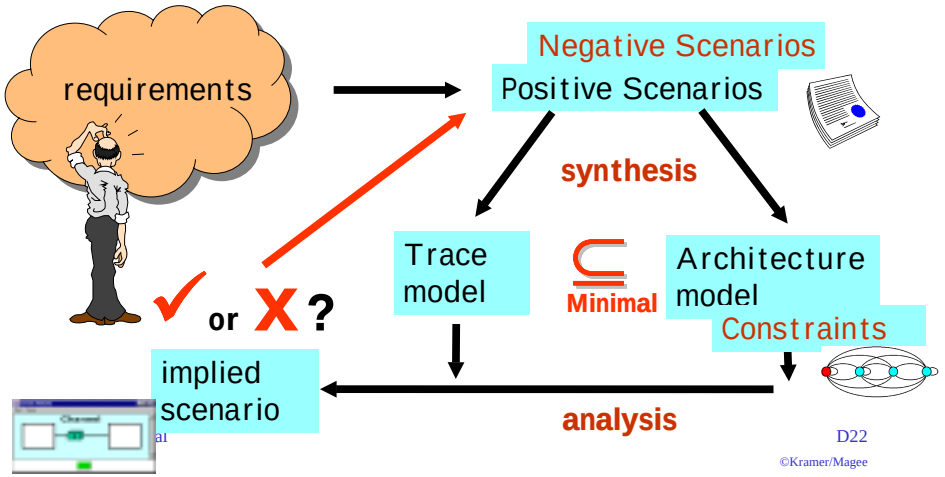
Negative Scenarios

- Basic Negative Scenarios
  - Allow push-button rejection
  - Reject 1 implied scenario at a time
  - Insufficient to allow process convergence
- Extended Negative Scenarios
  - Abstraction
  - Scope
  - Permit process convergence
  - Require "effort" from user.

c.f. [FSE'02]

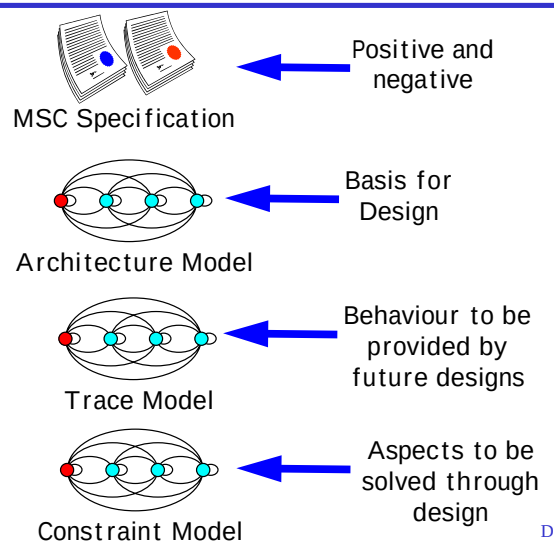
Synthesis and Elaboration

- Synthesis of the models, analysis by model checking, elaboration using implied scenarios

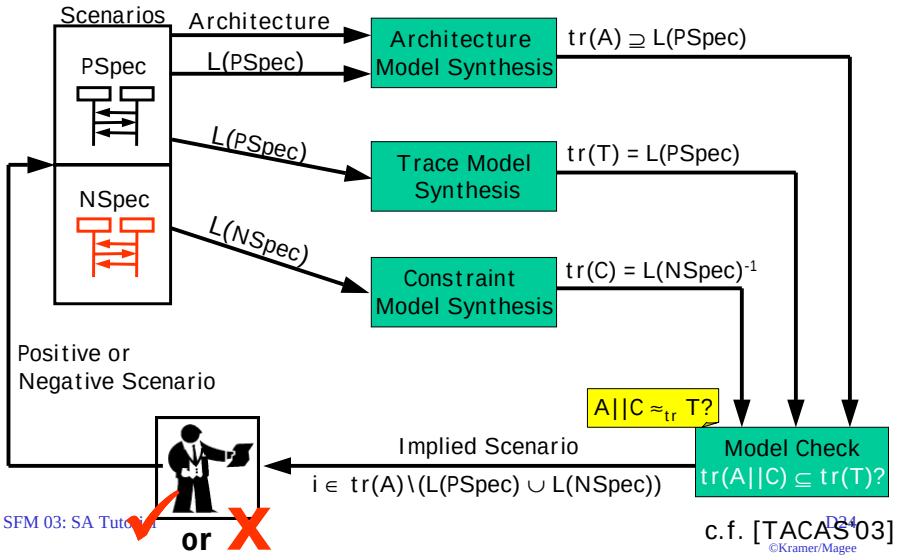


Outputs of the Elaboration Process

Elaborated to address the concurrent nature of the system



The Whole Picture



c.f. [TACAS'03]

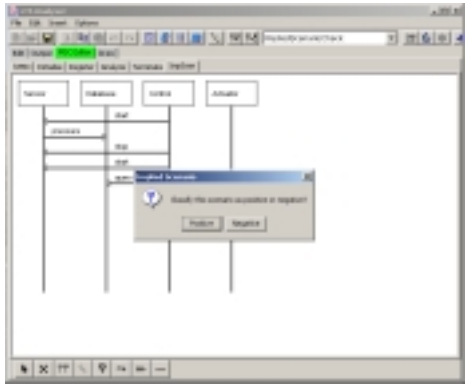
Summary

Implied scenarios…

- … are a mismatch between behaviour and architecture of MSC specifications.
- … can be detected using behaviour models.
- … indicate aspects of a MSC specification that should be further elaborated (what if….?).
- … can drive the elaboration of the MSC specification and behaviour models.

Tool support

- Implemented in Java.
- LTSA Plug-in
- Includes:
  - MSC Editor.
  - Negative and Positive Scenarios.
  - Synthesis of minimal architecture model.
  - Detection of implied scenarios.
  - Supports elaboration process.



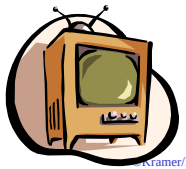
- Available at www.doc.ic.ac.uk/ltsa

Case Studies

- Railcar Transport System [Harel et al]

- B2B e-commerce site of greek industrial partners (STATUS project)

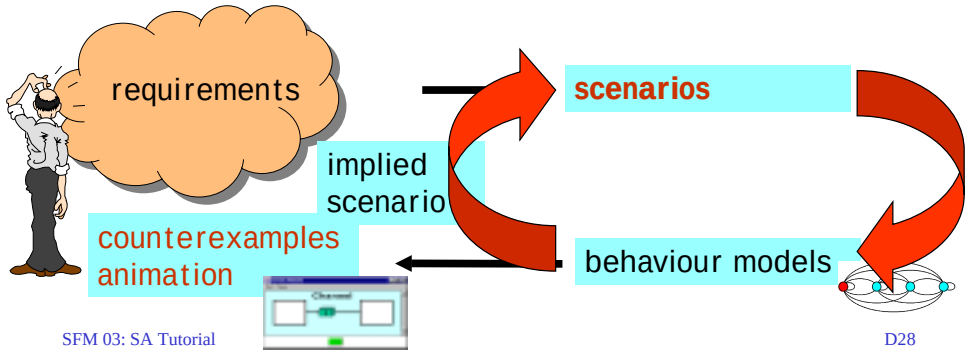
- Phillips Horizontal Communications Protocol for new product line of television sets.



scenarios to behaviour models … and back !

- **Use scenarios to aid in model construction.**
- **And vice-versa?**

  Use models to aid in scenario elaboration.



Related Work

- Term "Implied scenario" (for bMSCs only) [Alur et al., 2000].
- State-based MSC semantics [ITU 1996][…][…]….
- MSC expressiveness and Model Checking of MSCs [Peled et. al. 2001][Alur et. al 2001]
- Analysis
  - Implied scenarios [Muccini, 2001][Alur et al., 2001]
  - Non-local choice: [Leue et al. 1996].
  - Race conditions, pattern matching [Holzmann et al. 1996].
  - …
- Iterative completion of MSCs [Tarja et al. 2000].
- Negative scenarios: Live Sequence Charts [Harel et. al., 1999]
- Live Sequence Charts [Harel et al, 1999]
- Eliciting properties from scenarios: [Lamsweerde, 2000].

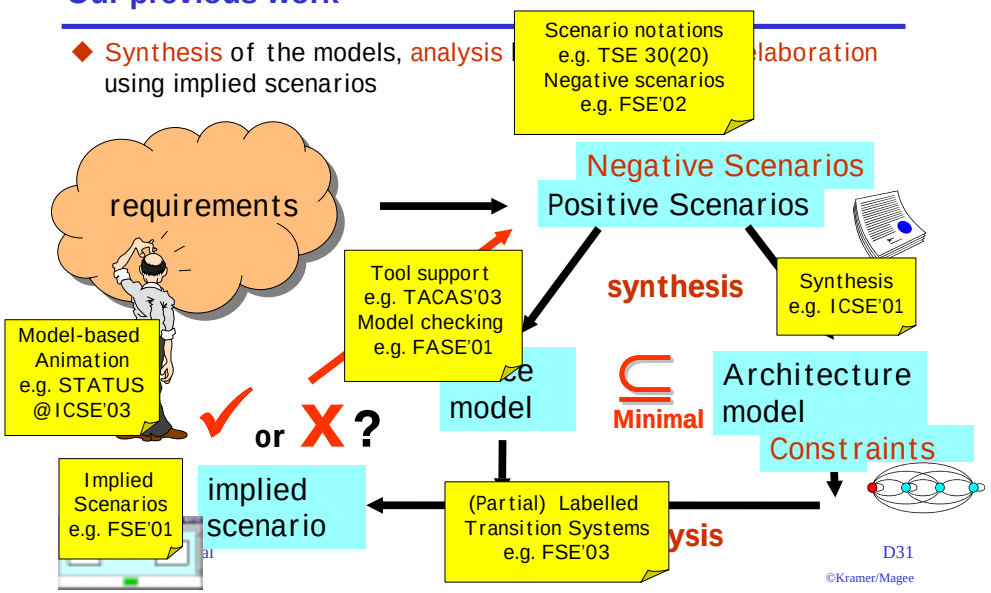
Our previous and ongoing work

- Scenario specifications (e.g. TSE 30(2))
- Synthesis (e.g. ICSE'01)
- Implied Scenarios (e.g. FSE'01)
- Negative Scenarios (e.g. FSE'02)
- Tool Support (e.g. TACAS'03)
- Model-based animation (e.g. STATUS@ICSE'03)
  - STATUS project with UPM, RoG, LogicDis
  - Play-in scenarios.
- Partial behaviour models (e.g. FSE'03)
  - Chechik (Toronto)

- Architecture + Scenarios = Model

Our previous work

- Synthesis of the models, analysis … laboration using implied scenarios



Models + Scenarios Further Reading

[1] Uchitel S., Kramer J. and Magee J., "Synthesis of Behavioral Models from Scenarios", *IEEE Trans. on Software Eng.*, SE-29 (2), (Feb. 2003), 99-115.

[2] Uchitel S. and Kramer J., "A Workbench for Synthesising Behaviour Models from Scenarios" *23rd IEEE/ACM Int. Conf. on Software Engineering (ICSE-2001)*, Toronto, Canada, May 2001, 188-197.

[3] Uchitel S., Kramer J. and Magee J., "Detecting Implied Scenarios in Message Sequence Chart Specifications", *9th ACM SIGSOFT Symposium on the Foundations of Software Engineering /8th European Software Engineering Conference (FSE / ESEC '2001)*, Vienna, September 2001,74-82.

[4] Uchitel S., Kramer J. and Magee J., "Negative Scenarios for Implied Scenario Elicitation", *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, Charleston, South Carolina, November 18-22, 2002.

[5] Uchitel S., Chatley R., Kramer J. and Magee J., "LTSA-MSC : Tool support for behavioral model elaboration using scenarios", *Tool Demo in 9th International Conference on Tools and Algorithms for the construction and Analysis of Systems (TACAS 2003)*, ETAPS 2003, Warsaw, Poland, April 2003.

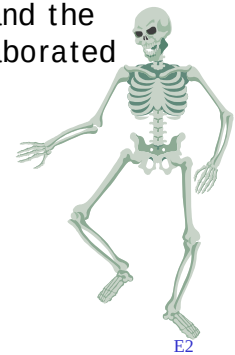
[6] Uchitel S., Kramer J. and Magee J., "Behaviour Model Elaboration using Partial Labelled Transition Systems", *4th joint ACM SIGSOFT Symposium on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '2003)*, Helsinki, Finland, September 2003.

[7] Chatley R., Kramer J., Magee J., and Uchitel S., "Model-based Simulation of Web Applications for Usability Assessment", *International Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, Portland, May 2003).

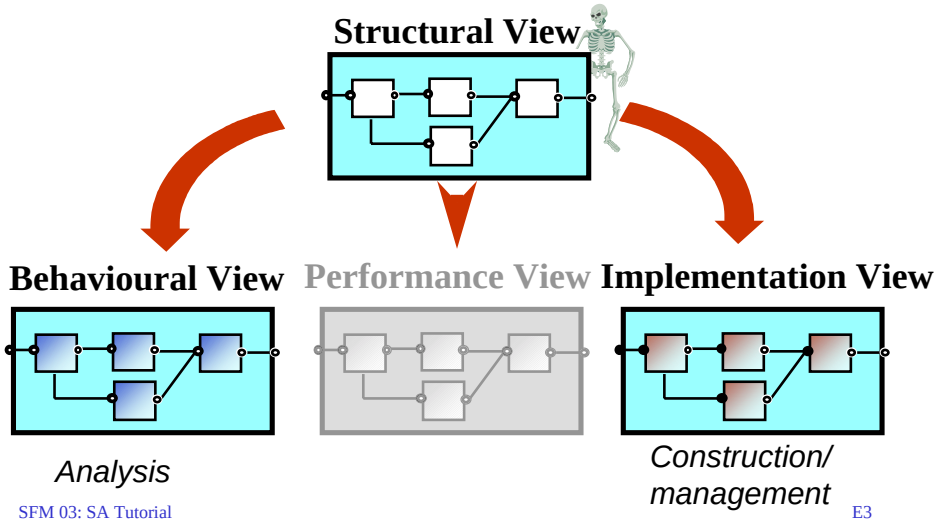
Part E :
Conclusion

The central role of design architecture

- Design architecture describes the gross organization and global structure of the system in terms of its constituent components.
- We consider that the models for analysis and the implementation should be considered as elaborated views of this basic design structure.



multi-view



Performance Models

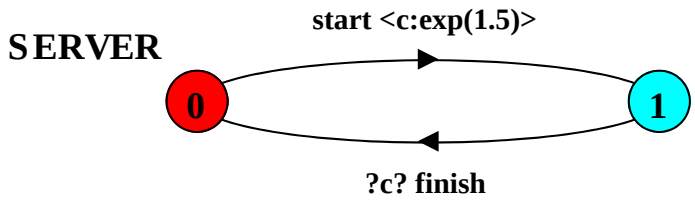
Approach is to augment FSP behavioural model with probabilistic choice and stochastic delay.

Associate discrete event simulation package with LTSA to extract results from the augmented model.

Use simulation rather than analytic methods so that we are not constrained as to distributions used for delays - however, plan to use analytic backend for Markov models.

Stochastic Delay

```
SERVER
  = (start<c:exp(MU)> ->?c?finish->SERVER).
```



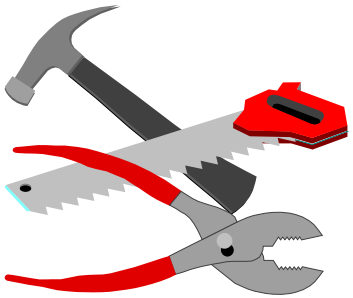
c = local clock

Scenarios + Architecture = Model ?

- A scenario describes an example of component instances interacting, in a particular architectural instance.
- An architecture describes the structure of the system in terms of its constituent components (instances and their types), with their interfaces and bindings, often in an hierarchy.
- Many systems include multiple component instances of the same type, and constrain interactions through the particular interfaces and bindings. Also components may be configured in a number of ways in an architecture.

How can we exploit this architectural information to improve our ability to build models?

Software tools - the need for automated support



Automated software tools are essential to support software engineers in the design process.

Techniques which are not amenable to automation are unlikely to survive in practice.

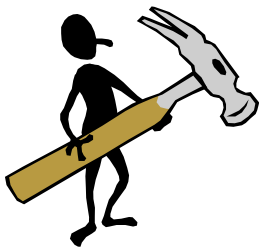
Experience in teaching the approach to both undergraduates and postgraduates in courses on Concurrency.
Initial experience with R&D teams in industry (BT, Philips)

Software Tools - Lightweight vs. Heavyweight

Short learning curve.
Immediate benefits.

VS.

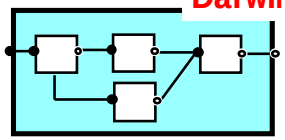
"PVS is a *large and complex* system and it takes a long while to learn to use it effectively. You should be prepared to invest six months to become a moderately skilled user."



Software Architecture Modeling & Analysis

Software Architecture describes gross organization of a system in terms of components and their interactions.

Darwin

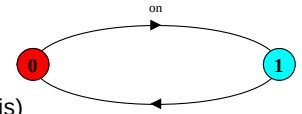


Behavior Modelling:

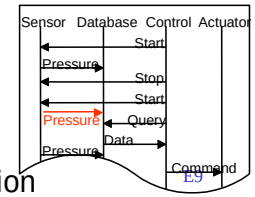
State Machines in form of LTS (Labelled Transition Systems)

Analysis using Model Checking CRA (Compositional Reachability Analysis) and LTL (Linear Temporal Logic)

Interpretation using animation



Requirements in the form of MSC (Message Sequence Charts) for model generation and elaboration



Software Architecture

Modeling & Analysis:

a rigorous approach

Jeff Kramer & Jeff Magee

Imperial College, London.