Formal modelling of resilient systems

Elena Troubitsyna

Åbo Akademi University,

Turku, Finland

Motivation

Dependability of a computing system is the ability to deliver a service that can be justifiably trusted.

Dependability attributes:
Availability, *reliability*, *safety*, *security, integrity, maintainability*

- *Main threat to dependability is complexity*

- Rich experience in modelling dependable systems from various domains in the FP7 EU Rodin and Deploy projects (8 years of experience)

- Contribution towards creating dependability-explicit development process

Motivation (cnt)

- Resilience is a further development of the dependability concept
- Resilience - the ability of a system to persistently deliver trustworthy services despite changes
- It encompasses the system aptitude to autonomously adapt to evolving requirements, operating environment changes and/or component failures

Structure

- Why formal engineering?
- Introduction into Event-B specification
  - Modelling and verifying safety
- Systems approach
- Refinement in Event-B
  - Fault tolerant control systems
- From models to safety cases
- Modelling in large
  - Layered architectures
  - Mode-rich systems
- Fault tolerant service-oriented systems
- Probabilistic extension
- Discussion

- How to demonstrate resilience?

Demonstrating resilience: traditional vs software engineering

- Build mathematical models of the design, its environment and requirements
- Use calculations to establish that the design in the context of the environment satisfies the requirements
- Modelling is validated by limited testing (because of continuous behaviour)
- It is product-based assurance

Demonstrating resilience: traditional vs software engineering

- According to probability theory demonstrating failure rate 10^{-n} /t requires 10^{n} tests
  - (Rushby: showing failure rate 10^{-9} requires 114000 years of testing)

- Even operational statistic is insufficient (for system working since 1993 without failures due to software, we can demonstrate probability of failure lower than 10^{-6}/h)

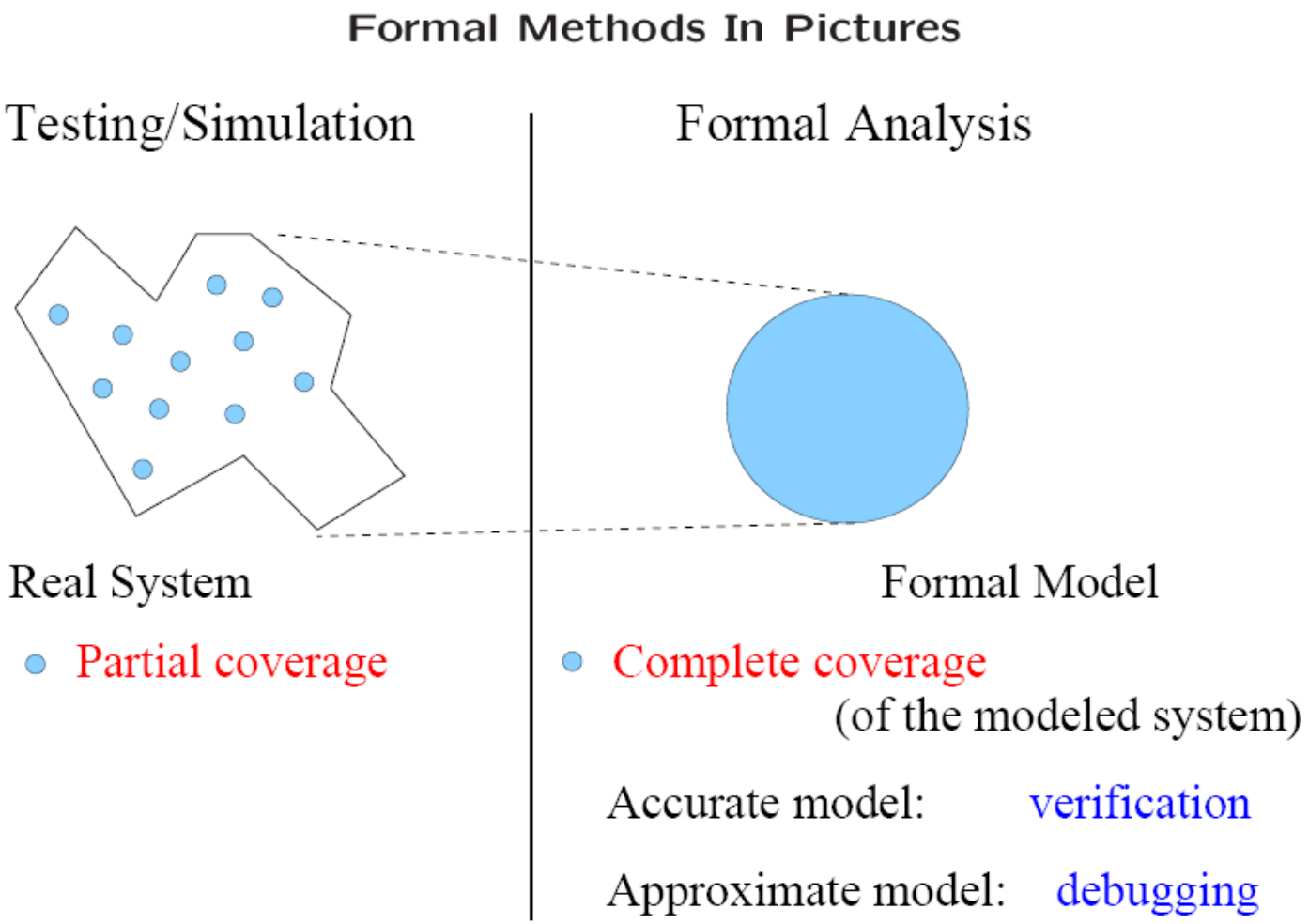
Demonstrating resilience: traditional vs software engineering

- Mostly done by controlling, monitoring, and documenting the process used to create SW

- Process-based assurance (i.e., no direct evidence about the product)

- Why: infeasibility of exhaustive testing
- Incomplete testing cannot be extrapolated (because of discrete behaviour)

Formal modelling: why



From J.Rushby talk on “Disappearing formal methods”

Demonstrating resilience: trends

- Trend 1: Emphasis on the development process aiming at producing fault free software

- Trend 2: System approach to demonstrate resilience

Formal modelling: why?

- To clean up architecture, handle complexity, facilitate verification
- To spot contradicting (and sometimes missing requirements)
- To clearly describe system static and dynamic properties

Historical note

- The B Method: invented in 1990-s by J.-R. Abrial to formally specify and develop sequential systems correct by construction;
- 1990-s: The Action Systems formalism by R.Back and K.Sere;
- from 2000: Event-B -- extension of the B Method in the spirit of Action Systems;
- from 2007: The Rodin Platform -- free industrial-strength tool support for Event-B
- Wide use of Event-B in the railway domain

Event B

- Specialisation of the B-Method
- Event B has been successfully used in development of several complex real-life applications
- It adopts top-down development paradigm based on refinement
- Refinement process: we start form an abstract formal specification and transform it into an implementable program by a number of correctness-preserving steps
  - It allows to structure complex requirements
  - Small transformations simplify verification
  - Verification by theorem proving does not lead to state explosion

Modelling in Event-B

- Overall system behaviour: a (potentially) infinite loop of system events:

forever do

Event1 or

Event2 or

Event3 or ...

end

- The dynamic system behaviour is described in terms of guarded commands (events):

  Stimulus → response.

Modelling in Event-B

- Overall system behaviour: a (potentially) infinite loop of system events:

forever do

Event1 or

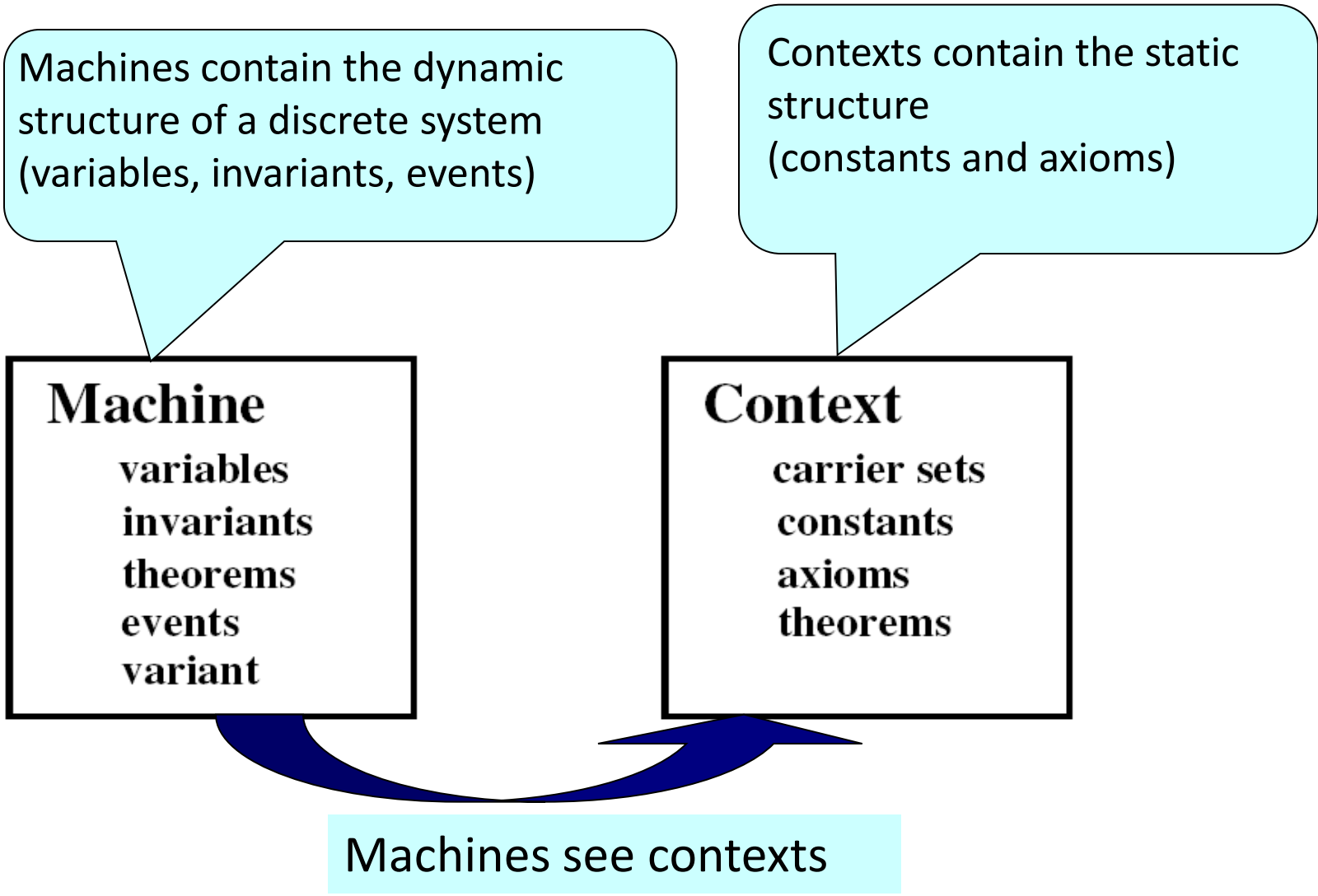
Event2 or

Event3 or ...

end

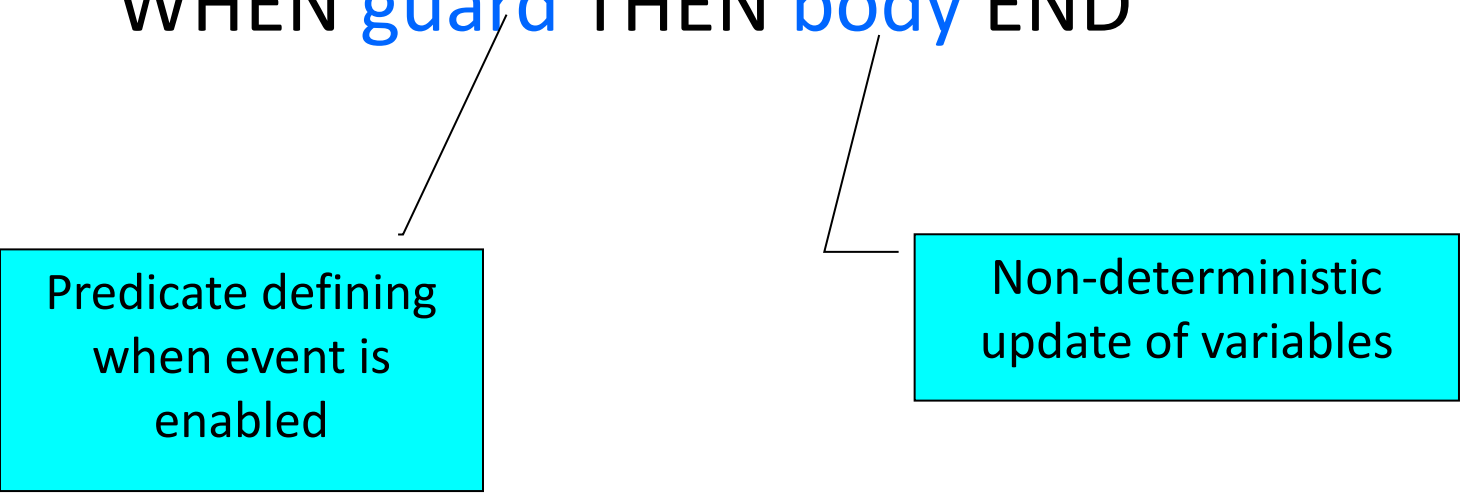
- Model invariant defines a set of allowed (safe) states;
- Each event should preserve the invariant;
  - We should verify this by proofs.

System Model in Event B

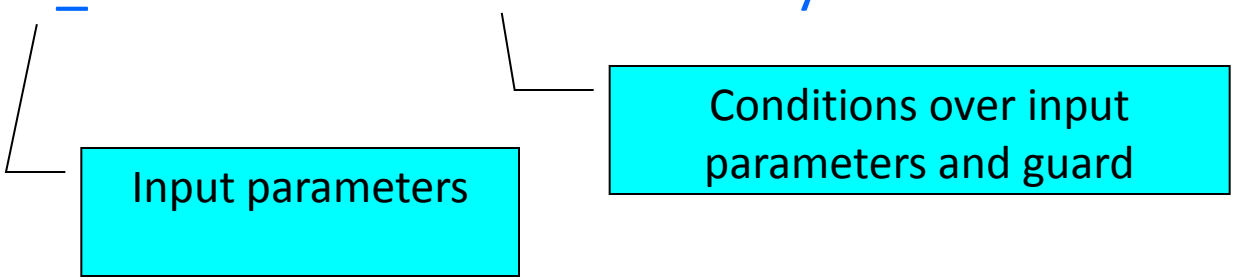


General form of event

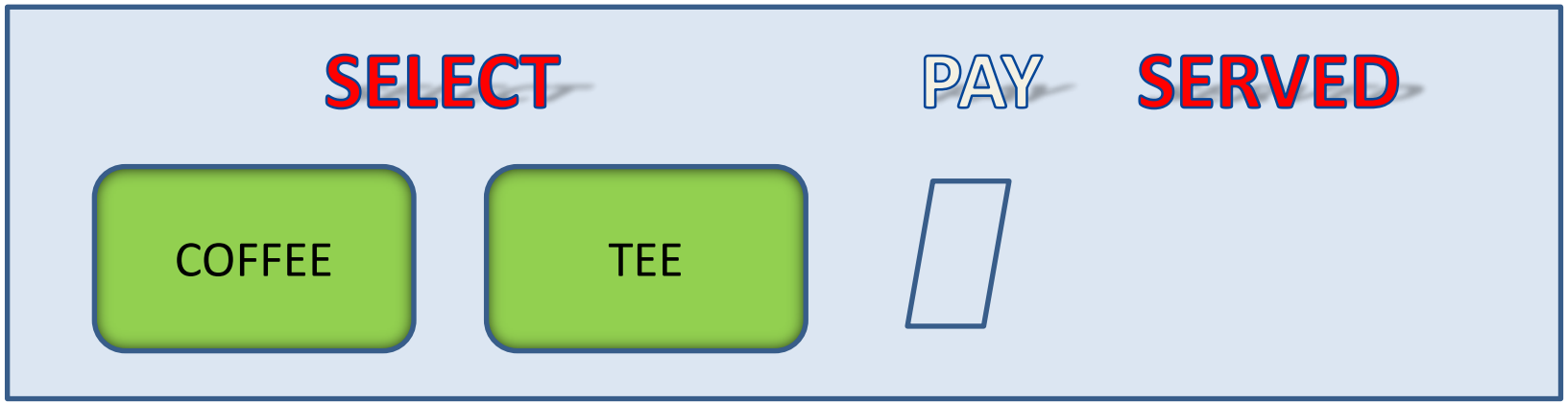
WHEN guard THEN body END



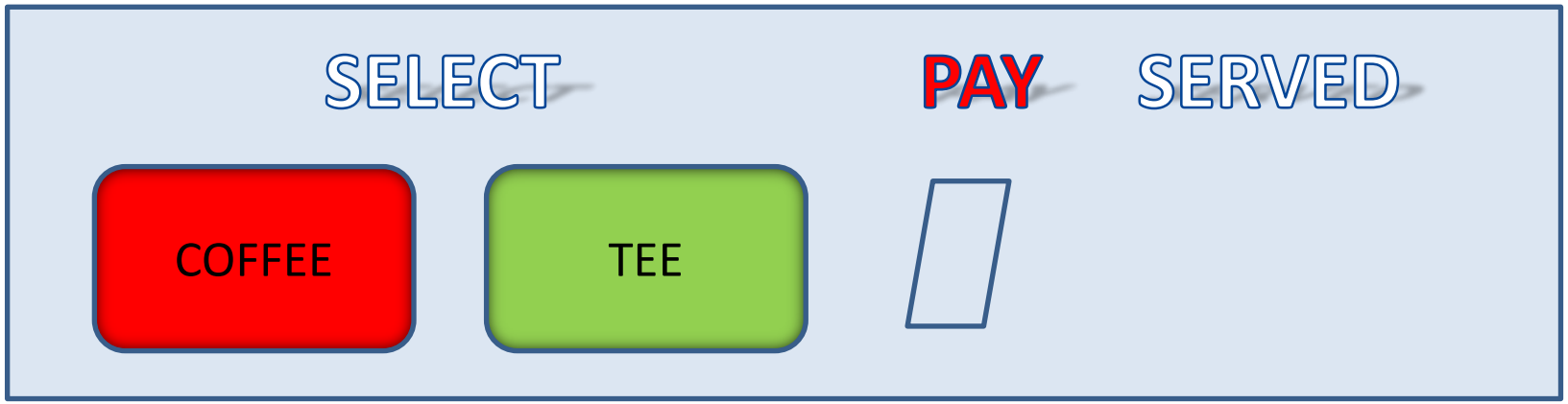
ANY local_var WHERE cond THEN body END



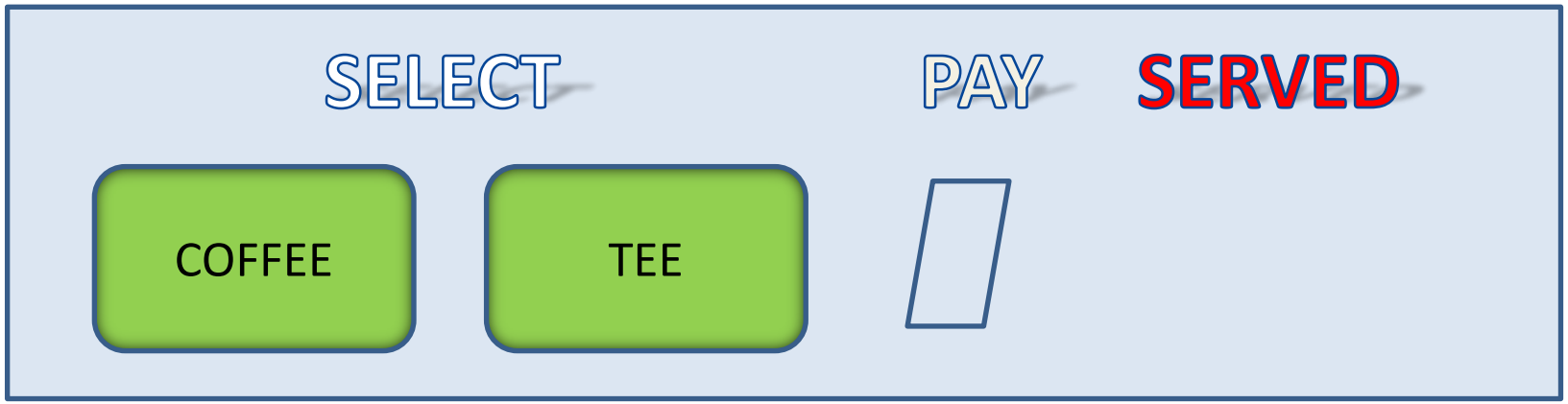
Tiny example: a vending machine



Tiny example: a vending machine



Tiny example: a vending machine



Tiny example: a vending machine

```
machine Main sees Data

variables selected payed served

events
  event INITIALISATION
    then
      @act1 selected ≔ NONE
      @act2 payed ≔ TRUE
      @act3 served ≔ TRUE
  end

  event select
    where
      @grd1 served = TRUE
    then
      @act1 selected :∈ {COFFEE,TEE}
      @act2 served ≔ FALSE
      @act3 payed ≔ FALSE
  end

  event pay
    where
      @grd1 selected ≠ NONE
    then
      @act1 payed ≔ TRUE
  end

  event serve
    where
      @grd1 payed = TRUE
    then
      @act1 selected ≔ NONE
      @act2 served ≔ TRUE
  end
```

Tiny example: a vending machine

```
machine Main sees Data

variables selected payed served

invariants
  @inv1 selected ∈ CHOICES
  @inv2 payed ∈ BOOL
  @inv3 served ∈ BOOL
  @inv4 served = TRUE ⇒ payed = TRUE
  @inv5 selected ≠ NONE ⇔ served = FALSE
```

where CHOICES is defined in the context component (**Data**) as an enumerated set $CHOICES = \{COFFEE, TEE, NONE\}$

Model verification

- Proof-based semantics: consider all possible executions at once;
- A model is converted into a number of proof obligations;
- A proof obligation is a mathematical theorem;
- Every proof obligation must be proven correct.

Model verification

- Verify that
  - Well-definedness conditions are satisfied
  - Initialization establishes invariant
  - Each event preserves invariant
- Verification is done by proofs
- Tool support – Rodin platform to generate and dischard proof obligations

Invariant preservation

- An invariant property is assumed to hold before every event;
- Each event must re-establish it:

$$I(s, c, v) \wedge G(s, c, v) \wedge R(s, c, v, v') \Rightarrow I(s, c, v')$$

where

- $I(s, c, v)$ – the model invariant;
- $G(s, c, v)$ – the event guard;
- $R(s, c, v, v')$ – the event action;
- v – old states;
- v' – new states.

The Rodin platform -- tool support for Event-B

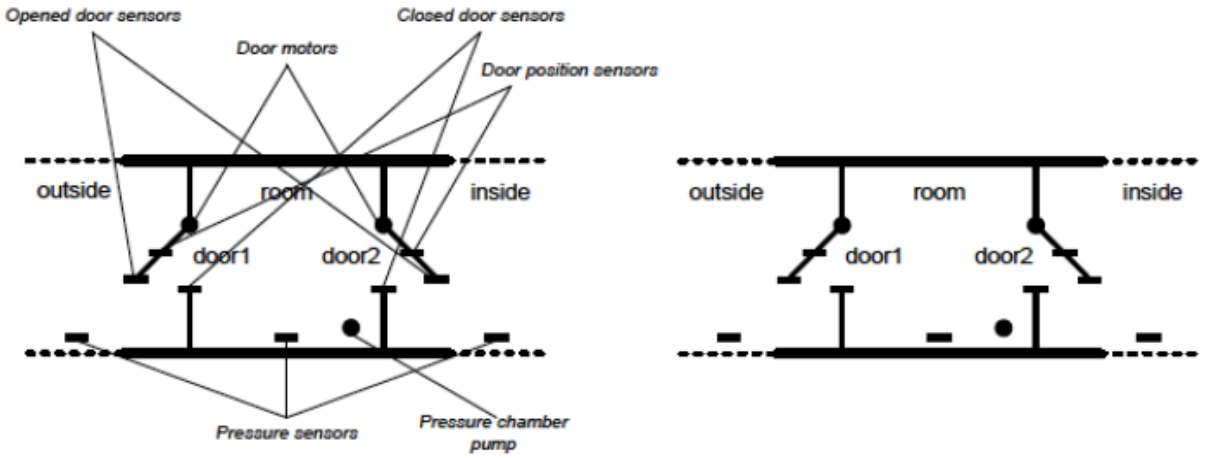
- Automates incremental development by refinement;
- Supports strong interplay between modelling and proof;
- Reactive: analysis tools are automatically invoked in the background whenever a change is made.
- The platform is extendable by plug-ins that
  - extend the Event-B language and proving techniques;
  - bridge the platform with various model-checkers, theorem provers, animators, modelling notations (e.g., UML), etc.

Resilience explicit modelling: safety

- Safety is a property of a system to not endanger human life or environment

- Safety requirements are represented either as invariants or reflected in the event guards (restrict when an event can be executed)

Example



- Sluice connects areas with dramatically different pressures;
- It is unsafe to open a door unless the pressure is levelled between the connected areas;
- The purpose of the system is to operate doors safely by adjusting the pressure in the room.

Defining invariant

```
invariants
  @inv1 door1 ∈ DOOR // current status of door1
  @inv2 door2 ∈ DOOR // current status of door2
  @inv3 pressure ∈ PRESSURE // current status of system pressure
  @inv4 ¬ (door1 = OPEN ∧ door2 = OPEN)
```

Modelling safety

```
invariants
  @inv1 door1 ∈ DOOR // current status of door1
  @inv2 door2 ∈ DOOR // current status of door2
  @inv3 pressure ∈ PRESSURE // current status of system pressure
  @inv4 ¬ (door1 = OPEN ∧ door2 = OPEN)
  @inv5 door1 = OPEN ⇒ pressure = HIGH
  @inv6 door2 = OPEN ⇒ pressure = LOW
```

```
machine m0 // initial abstract model of the sluice gate system
 sees ctx0

variables door1 door2 pressure
 event INITIALISATION
   then
     @act1 door1 ≔ CLOSED
     @act2 door2 ≔ CLOSED
     @act3 pressure :∈ PRES
 end

 event open1
   where
     @grd1 door1 = CLOSED
     @grd2 door2 = CLOSED
   then
     @act1 door1 ≔ OPEN
 end

 event close1
   where
     @grd1 door1 = OPEN
   then
     @act1 door1 ≔ CLOSED
 end
```

```
event open2
  where
    @grd1 door1 = CLOSED
    @grd2 door2 = CLOSED
  then
    @act1 door2 ≔ OPEN


event pressure_low
  where
    @grd1 door1 = CLOSED
    @grd2 door2 = CLOSED
  then
    @act1 pressure ≔ LOW
end
```

```
machine m0 // initial abstract model of the sluice gate system
 sees ctx0

variables door1 door2 pressure
 event INITIALISATION
   then
     @act1 door1 ≔ CLOSED
     @act2 door2 ≔ CLOSED
     @act3 pressure :∈ PRES
 end

 event open1
   where
     @grd1 door1 = CLOSED
     @grd2 door2 = CLOSED
     @grd3 pressure = HIGH
   then
     @act1 door1 ≔ OPEN
 end

 event close1
   where
     @grd1 door1 = OPEN
   then
     @act1 door1 ≔ CLOSED
 end
```

```
event open2
  where
    @grd1 door1 = CLOSED
    @grd2 door2 = CLOSED
    @grd3 pressure = LOW
  then
    @act1 door2 ≔ OPEN


event pressure_low
  where
    @grd1 door1 = CLOSED
    @grd2 door2 = CLOSED
  then
    @act1 pressure ≔ LOW
end
```

Modelling safety

```
invariants
  @inv1 door1 ∈ DOOR // current status of door1
  @inv2 door2 ∈ DOOR // current status of door2
  @inv3 pressure ∈ PRESSURE // current status of system pressure
  @inv4 ¬ (door1 = OPEN ∧ door2 = OPEN)
  @inv5 door1 = OPEN ⇒ pressure = HIGH
  @inv6 door2 = OPEN ⇒ pressure = LOW
```

- Formal verification helps us to ensure that no essential safety requirements are missed:
  - Invariant contains definition of safety
  - Failed proofs : strengthening guards of events

Refinement: informally

- Intuition: gradual elaboration on the system behaviour and data structures;
- A top-down development of a system;
- Refined behaviours and data structures should be consistent (non-contradictory) with more abstract ones;
- Helps to structure the system requirements;
- A way of handling of the system complexity and structuring of the proof effort;

The notion of model refinement

- Essential property: transitivity. Allows us to build a refinement chain of gradual development (unfolding) of the system;
- Mathematically:

$$\frac{M_1 \sqsubseteq M_2 \sqsubseteq .... \sqsubseteq M_n}{M_1 \sqsubseteq M_n}$$

- All proven properties of more abstract models are preserved by refinement;
- Many formalisations based on the idea of model refinement, e.g., Refinement Calculus, the B Method, ...

Refinement in Event-B

- Defined separately for a context and a machine;
- For a context component, it is called extension;
- Context extension allows
  - introducing new data structures (sets and constants), as well as
  - adding more constraints (axioms) for already defined ones.

Refinement in Event-B (cnt.)

For a machine component, there are several possible kinds of refinement:

- simple extension of an abstract model by new variables and events (superposition refinement);
- constraining the behaviour of an abstract model (refinement by reducing model non-determinism);
- replacing some abstract variables by their concrete counterparts (data refinement);
- a mixture of those

Superposition refinement

- Adding new variables and events;
- Reading and updating new variables in old event guards and actions;
- Interrelating new and old variables by new invariants;
- Restriction: the old variables cannot be updated in new events!

Refinement of non-determinism

- Focuses on the old (abstract) model events:
- Strengthening the guards;
- Providing several versions of the same event;
- Refining non-deterministic actions (assignments).

evt =
WHERE g **THEN**
$detected :\in BOOL$
END

evt1 **refines** *evt* =
WHERE $g \wedge g'$ **THEN**
$detected := TRUE$
END

evt2 **refines** *evt* =
WHERE $g \wedge g''$ **THEN**
$detected := FALSE$
END

Data refinement

- Replacing some old variables by their concrete counterparts;

- A part of concrete invariant, gluing invariant, describes the logical relationships between the old and new variables
  - The gluing invariant is used in all proofs to show the correctness of

$$(comm\_failure = TRUE) \Leftrightarrow (msg\_sent = \text{FALSE} \vee (msg\_sent = \text{TRUE} \wedge msg\_lost = \text{TRUE}))$$

Refinement proof obligations

- As an abstract model, a refined model should satisfy the feasibility and invariant preservation properties;
- In addition, we should show that
  - guards of the old events are strengthened (or remain the same);
  - actions of the old events simulate those of the abstract ones – each refined model transition (execution step) is allowed by the abstract model;
  - In all POs, the gluing invariant is used to relate the old and new model states.

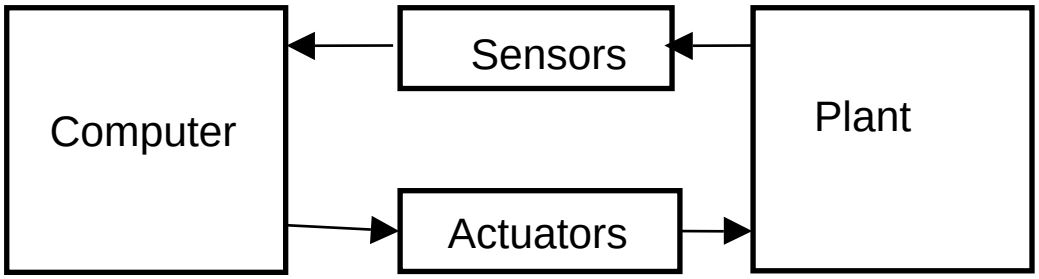
Systems approach

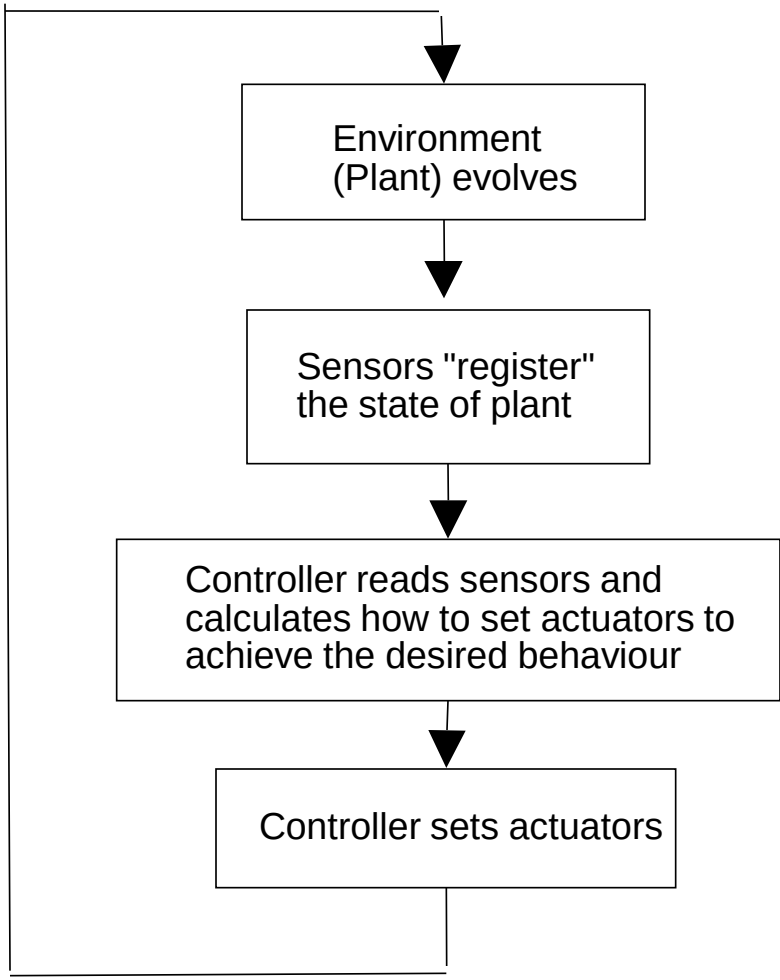
- System approach assumes that while developing SW we have a picture of whole system in mind

- Software fault
  - "Bug" -- bad implementation of good requirements
  - Design fault -- good implementation of bad requirements

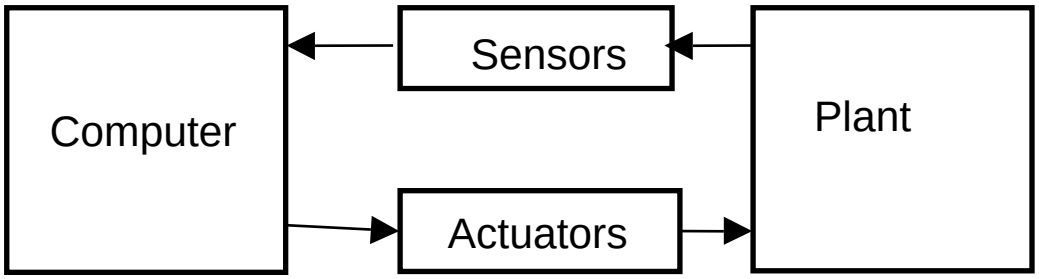
- We cannot obtain "good" requirements if we do not understand how the whole system works (and fails)

Systems Approach in Control System Modelling

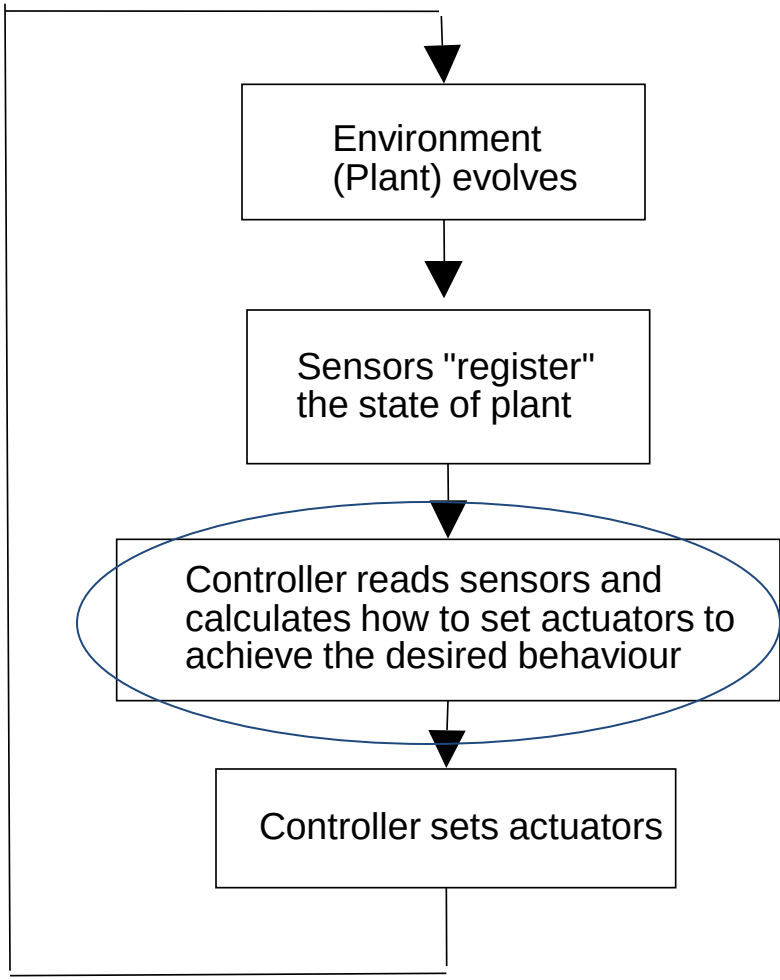




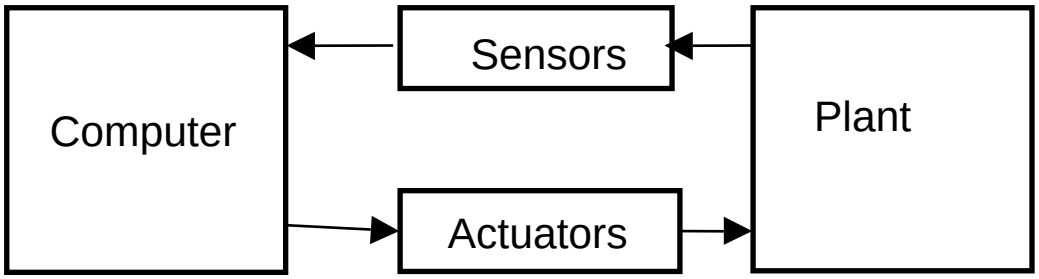
Systems Approach in Control System Modelling



Traditional development :
focus on controller (SW)

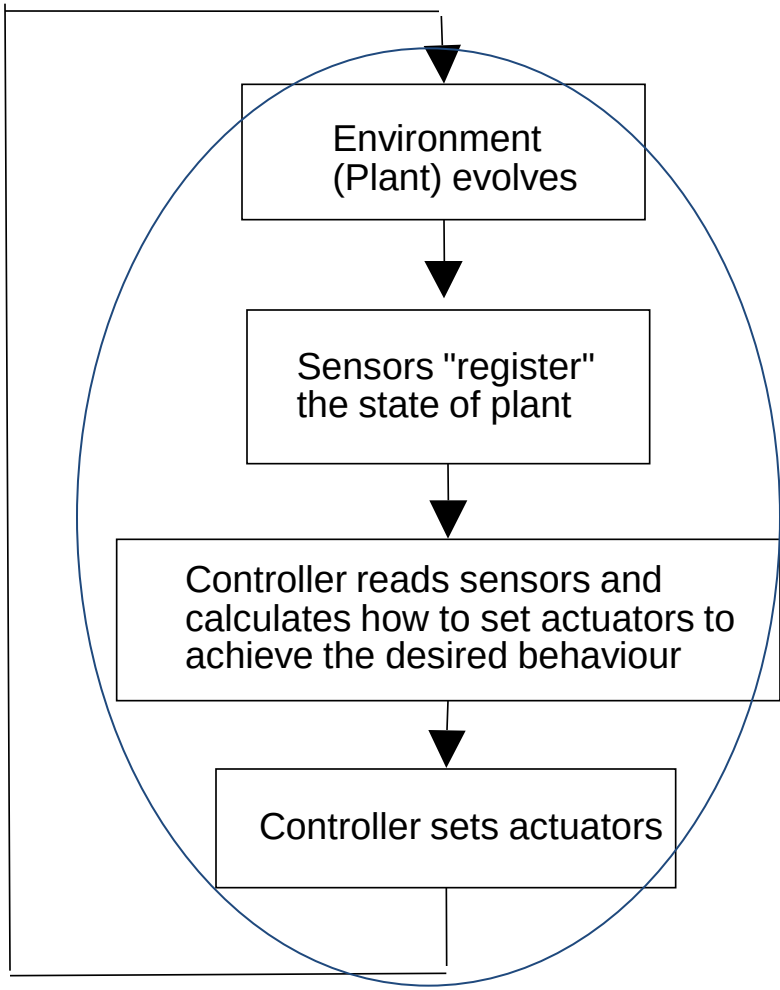


Systems Approach in Control System Modelling



Traditional development :
focus on controller (SW)

Systems approach:
model entire system and
derive controlling SW by
refinement and decomposition

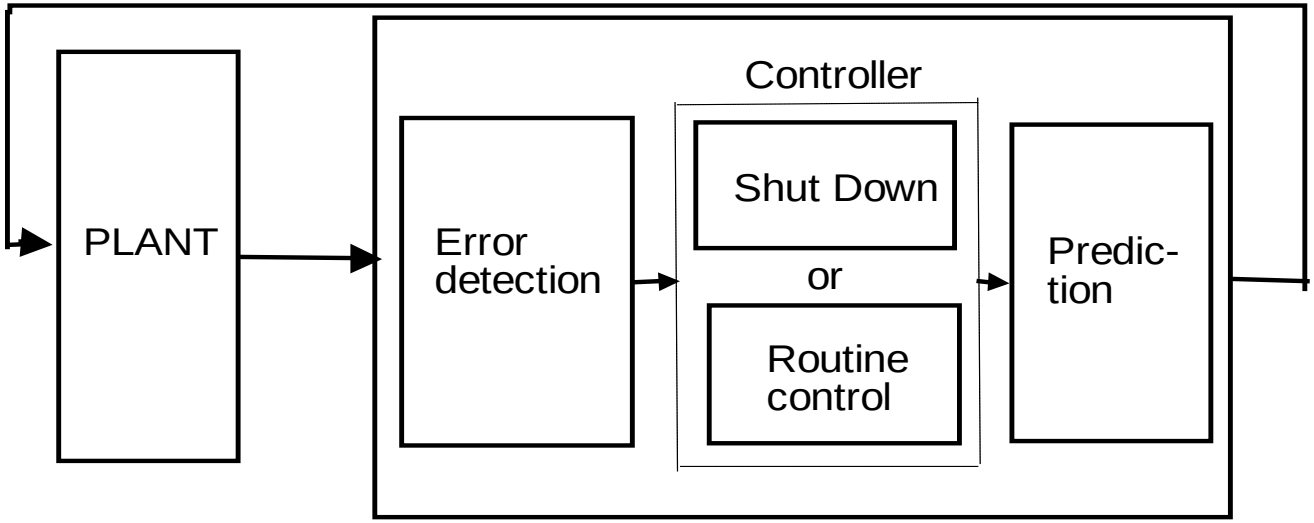


Modelling resilience: fault tolerance

- Fault tolerance is an ability of system to exhibit predictable behaviour in presence of faults
  - Fault masking
  - Error recovery
- Systems approach is essential: allows us to model failure occurrence and error detection
- Relies on the model of environment

Fault tolerant explicit modelling of control system

- Our specification of control system includes both a plant and a controller
- The overall behaviour of the system is an alternation between the events modelling plant evolution and controller reaction



MACHINE

ControlSystem

INVARIANT

safety invariant and types of variables

OPERATIONS

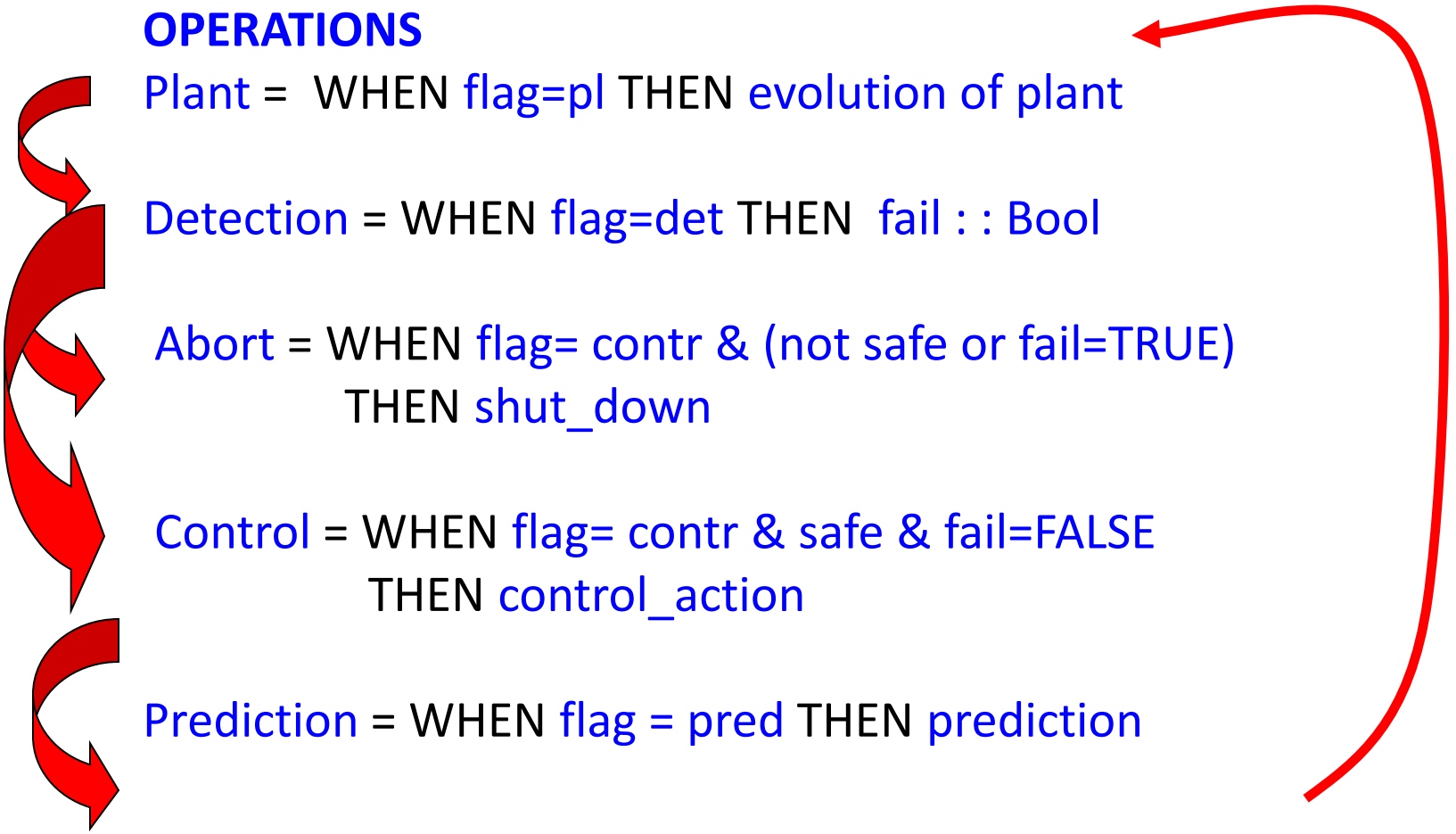
Plant = WHEN flag=pl THEN evolution of plant

Detection = WHEN flag=det THEN fail : : Bool

Abort = WHEN flag= contr & (not safe or fail=TRUE)
THEN shut_down

Control = WHEN flag= contr & safe & fail=FALSE
THEN control_action

Prediction = WHEN flag = pred THEN prediction



Example: abstract specification of heater controller

INVARIANT

…(fail=FALSE & flag/=DET & flag/=CONT => temp<=t_crit)

OPERATIONS

plant = WHEN flag = PL THEN temp :: NAT1 || flag := DET

detection = WHEN flag = DET THEN flag := CONT || fail :: BOOL

abort_op =
WHEN flag = CONT & (fail = TRUE or temp > t_crit) THEN abort

switch1 =
 WHEN flag = CONT & fail= FALSE & temp <= t_crit & temp < t_low
       THEN heat := ON…

prediction = WHEN flag = PRED THEN flag := PL END

Refinement of error detection

The basic mechanism:

- Simulate dynamics of fault-free and faulty plant in the plant operation
- Use dynamics of fault-free plant to calculate expected states
- Mismatch between the expected and observed states signals about error occurrence

Corresponding refinement:

- Plant: mathematical functions to model fault free dynamics and non-deterministic “deviations” from these functions to model random error occurrence
- Prediction: assignment to variables modelling expected values using functions modelling fault-free dynamics
- Detection: assignment to fail the result of matching obtained input values and expected ones

Refined plant of the heater

If heater is on and OK or it stuck at On then use math. functions min_incr and max_incr to calculate interval of sensed temperature

WHEN (heat = ON & heater_fail_sim = OK) or heater_fail_sim=ON_STUCK
THEN
temp :: min_incr(temp)..max_incr(temp)
…

If heater is OK but sensor failed then sensed temperature is outside of valid interval

WHEN (heat = ON & heater_fail_sim = Failed)

THEN temp :: {xx | xx:NAT1 & (xx<next_temp_min or xx>next_temp_max)}

Refinement of error detection via data refinement

- We replace variable fail modelling error occurrence by variables representing failures of system components
- In our example its either sensor or switch failures

sensor_fail : BOOL & heater_fail : H_FAIL

(fail=TRUE) <=> (sensor_fail=TRUE or heater_fail /= OK)

Refinement of error detection via data refinement (cont.)

Refinement of error detection mechanism

- Prediction: include calculations of expected states by using mathematical functions modelling fault free behaviour

```
Prediction1 =
 WHEN flag = PRED & heat = ON THEN
   next_temp_max,next_temp_min :=max_incr(temp),min_incr(temp)
 Prediction2 =
WHEN flag = PRED & heat = OFF THEN
    next_temp_max,next_temp_min := min_decr(temp),max_decr(temp)
```

Failure Modes and Effect Analysis

FMEA is a well-known inductive safety analysis technique

For each system component it defines its possible failure modes, local and system effect of component failures, as well as detection and recovery procedures.

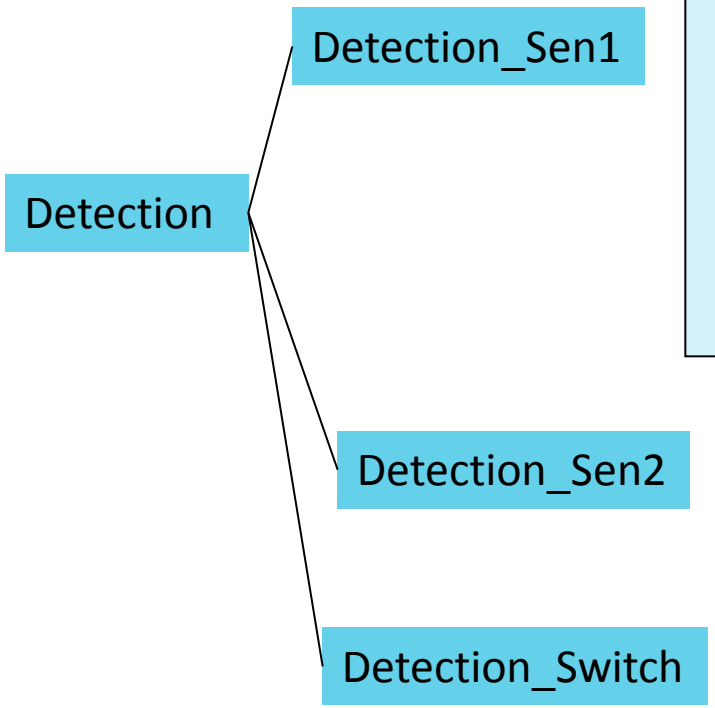
FMEA table fields

Component – name of a component
Failure mode – possible failure modes
Possible cause – possible cause of a failure
Local effects – caused changes in the component behaviour
System effect – caused changes in the system behaviour
Detection – determination of the failure
Remedial action – actions to tolerate the failure

Example of FMEA

Comp.	Sensor
Failure mode	Sensor reading exceeds expected range
Possible cause	Physical failure
Local effects	Sensor reading is out of expected boundaries
System effects	Potentially unsafe behaviour
Detection	Comparison of the value received with the expected
Remedial action	Retry three times. If failure persists then switch to redundant sensor, diagnose switch failure. If failure still persists, shut down and raise the alarm.

From FMEA to formal specification

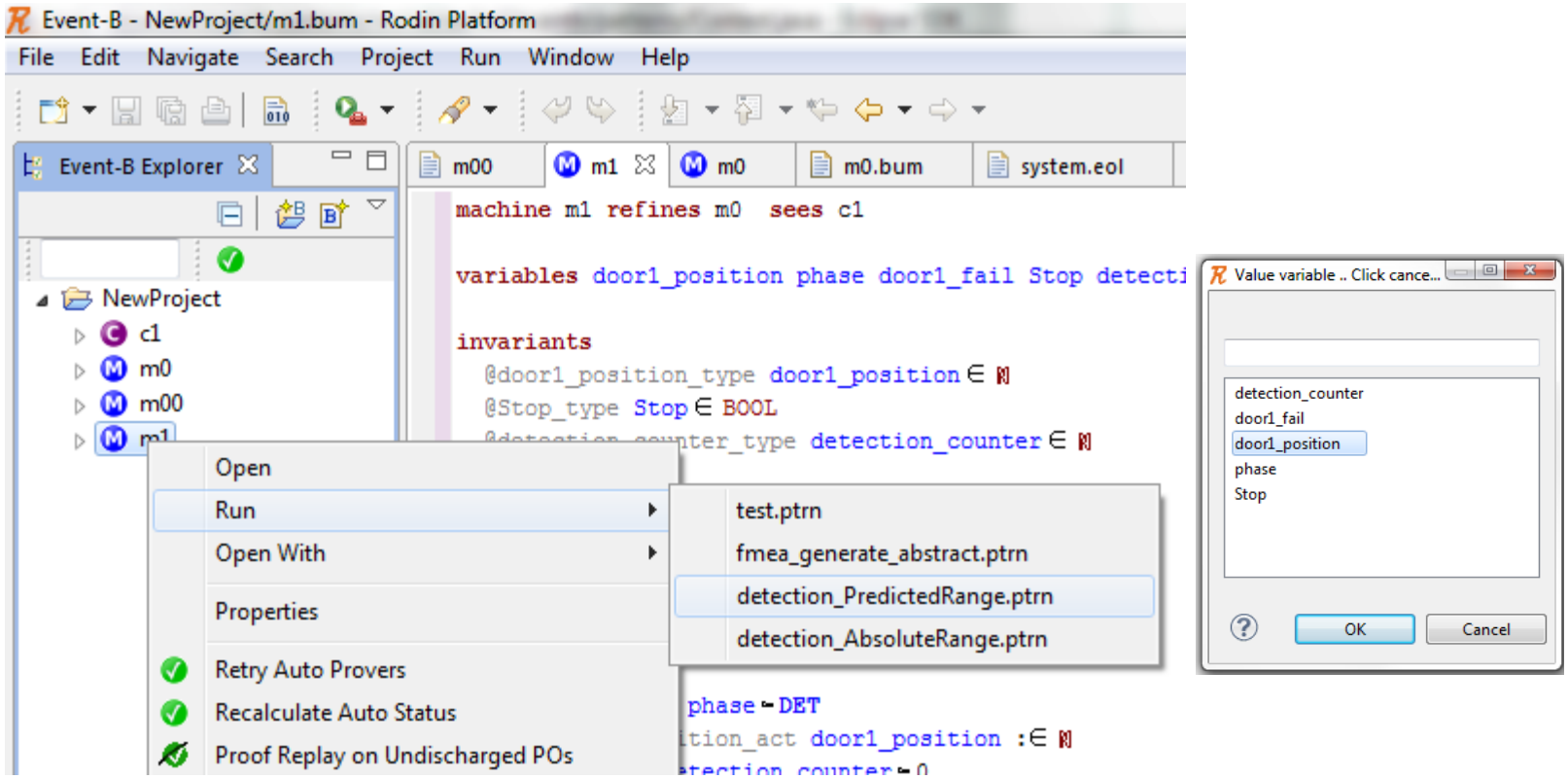


Checking conditions of detecting Failure Mode1 and changing FM1detected flag

Checking conditions of detecting Failure Mode2 and changing FM2 detected flag

...

Patterns application plug-in (Ilya Lopatkin, Newcaslte Univ)



Refined specification: general form

VARIABLES

state_variables of ControlSystem
new variables for modelling failures of components
variables modelling expected states

INVARIANT

constraints of variables & data refinement relation
…

EVENTS

Plant =
WHEN flag=pl
THEN simulation of evolution of the plant based on the corresponding physical laws and non-deterministic occurrence of failures
END;

Refined specification : general form (cont.)

Detection =
WHEN flag=det and real state does not match expected state
THEN failures of components are detected … END;

Abort =
WHEN flag= contr & (not_refined_safe \/ components failed) THEN abort END;

Control =
WHEN flag= contr & refined_safe & components are fault-free
THEN controlling_action … END;

Prediction =
WHEN flag = pred
THEN calculate next expected state using
the same physical laws as for simulating the plant …END

Introducing redundancy by refinement

- In systems without redundancy we can detect errors but cannot identify their causes
- Hence all errors have the same (high) criticality
- To distinguish between errors we need to employ redundancy
- It will allow us to split the set of faulty system states into the subset of faulty but safe (and hence operational) states and the subset of faulty and unsafe states.

General specification of system with redundancy

VARIABLES

state_variables of ControlSystem

new variables for modelling redundancy

variables modelling expected states

EVENTS

Plant =

WHEN flag=pl

THEN simulation of the behaviour of the plant with redundant components and the occurrence of component's failures; … END;

General specification of system with redundancy (cont.)

Detection 1=

WHEN flag=det and mismatch between the states of redundant components is detected & the error cannot be masked

THEN critical failure of redundant components is detected

END

Detection 2

WHEN the real state does not match the expected state

THEN critical failure of other components is detected

END;

…

General specification of system with redundancy (cont.)

Observe, that although the guard of abort did not change, it becomes enabled less often. (Because not-critical failures are now masked)

Abort =
WHEN flag= contr & (not refined_safe \/ components failed)
THEN abort END;

Control =
WHEN flag= contr & refined_safe &
components are fault free or failures are masked
THEN controlling_action; …END;

Prediction … is not affected by this refinement

Specifying redundant sensors

INVARIANT

…sen1 : NAT1 & sen2 : NAT1 & sen3 : NAT1 &

Temperature at the previous refinement step coincides with at least one sensor reading at the current refinement step

(temp=sen1 or temp=sen2 or temp=sen3) &

If at least two sensors produce identical readings, majority view is taken. Temperature at the previous refinement step coincides with majority view

(sen1=sen2 => temp=sen1) &

(sen2=sen3 => temp=sen2) &

(sen3=sen1 => temp=sen3) & …

Error detection via voting

detection =
WHEN flag = DET
THEN

If majority can be established then take majority view as current temperature and consider sensors to be fault free

WHEN sen1 = sen2 THEN temp1,sensor_fail := sen1,FALSE END
WHEN sen2 = sen3 THEN temp1,sensor_fail := sen2,FALSE END
WHEN sen3 = sen1 THEN temp1,sensor_fail := sen3,FALSE END

If majority CANNOT be established then current temperature coincides with nondeterministically chosen sensors, sensors failure is detected

WHEN sen1 /= sen2 and sen2/= sen3 and sen1 /= sen3
THEN temp1 :: {sen1,sen2,sen3}; sensor_fail := TRUE END;

Outline of the approach

1. Abstract specification of entire system: the initial specification captures requirements for routine control, models failure occurrence and defines safety property as a part of its invariant

2. Specification with refined error detection *mechanism*: the abstract specification is augmented with the representation of failures of the components, more elaborated description of plant's dynamics and detailed description of error detection.

Outline of the approach

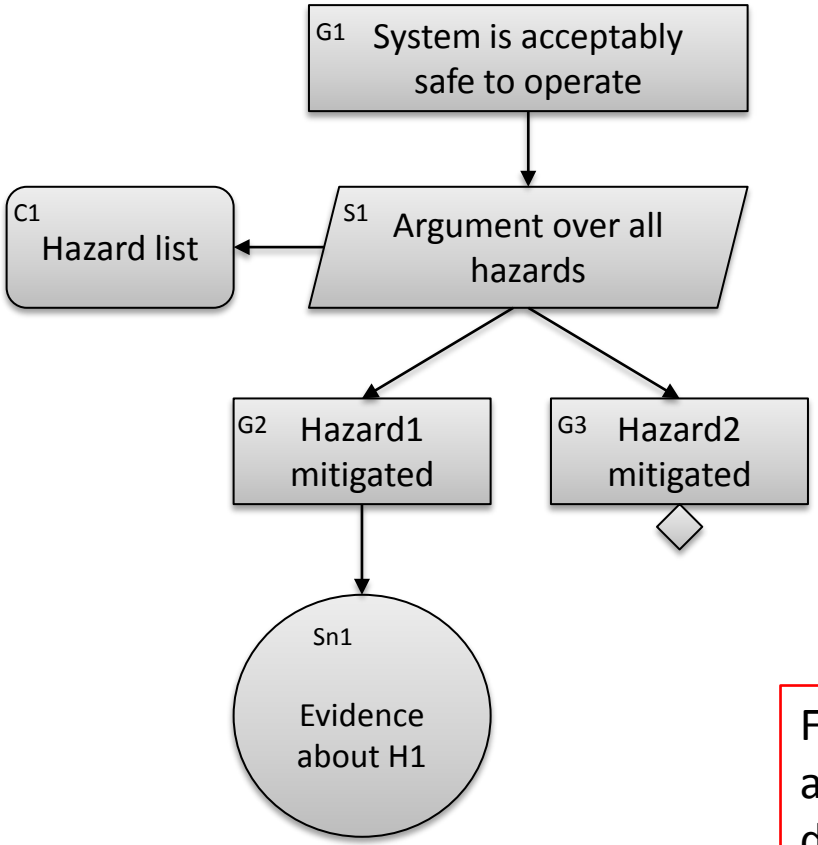
3. Specification of the system supplemented with redundancy: the specification is refined to describe behaviour of redundant components and control over them. The error detection mechanism is enhanced to distinguish between criticality of failures.
4. Decomposition: the specification of overall system is split into specifications of the controller and the plant.
5. Implementation: executable code of controller is produced.

Formal modelling and certification of safety-critical systems

- IEC 61508: four safety integrity levels (SILs)
  - SIL 3 requires formal modelling
  - SIL 4 requires formal verification

- Safety Case – a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.
  - Goal -> Strategy -> Argument

Example of a Safety Case



- What can be used as an evidence?
  - PHA, FTA, etc.
  - Checks done by a reviewer (an expert)
  - Tests
  - Model checking results
  - Proofs

Formal proofs as the evidence for safety cases are reasonable if those proofs are demonstrated to support incorporated safety requirements

Linking Event-B and Safety Cases

Goals

- Usually goals correspond to safety requirements
  - All target requirements should be represented in the model

Argument

- Technique showing how the goal is achieved
  - Each requirement should be verified

We propose

- Taxonomy of requirements
- Define how they should be reflected in the model
- Define verification means

Modelling in Large

- Event-B is a language for system-level modelling
- We can start with an abstract model at architectural level, refine it, decompose and formally develop lower level components
- Modelling functional behavious and fault tolerance at different abstraction levels

Architectural modellling

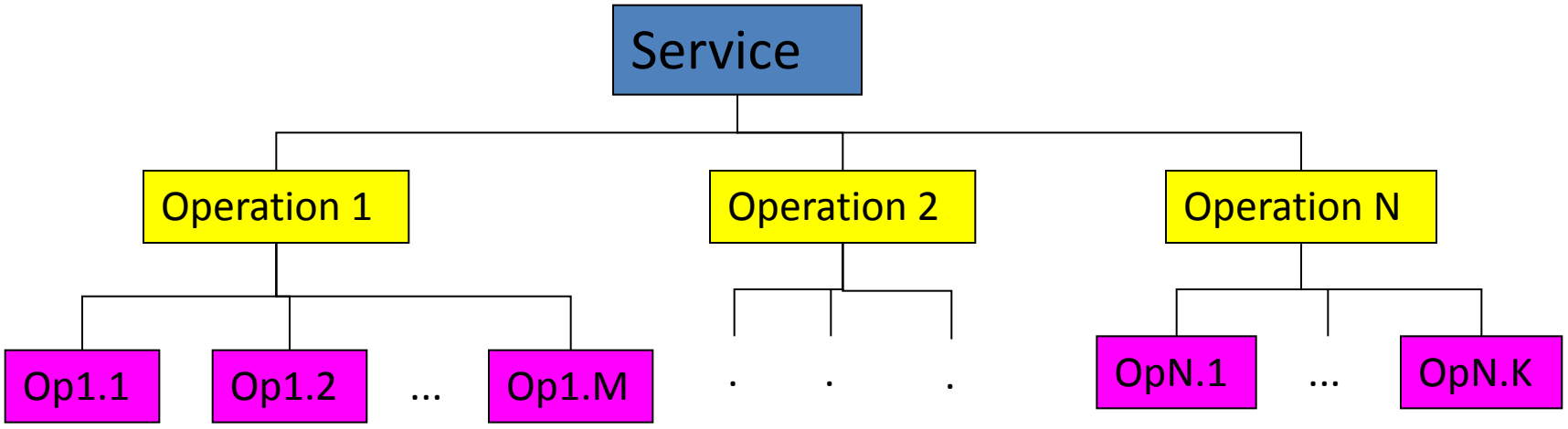
- Fault tolerant control systems with layered architecture
- Mode-rich systems

Deriving architecture by refinement: pharmaceutical robot

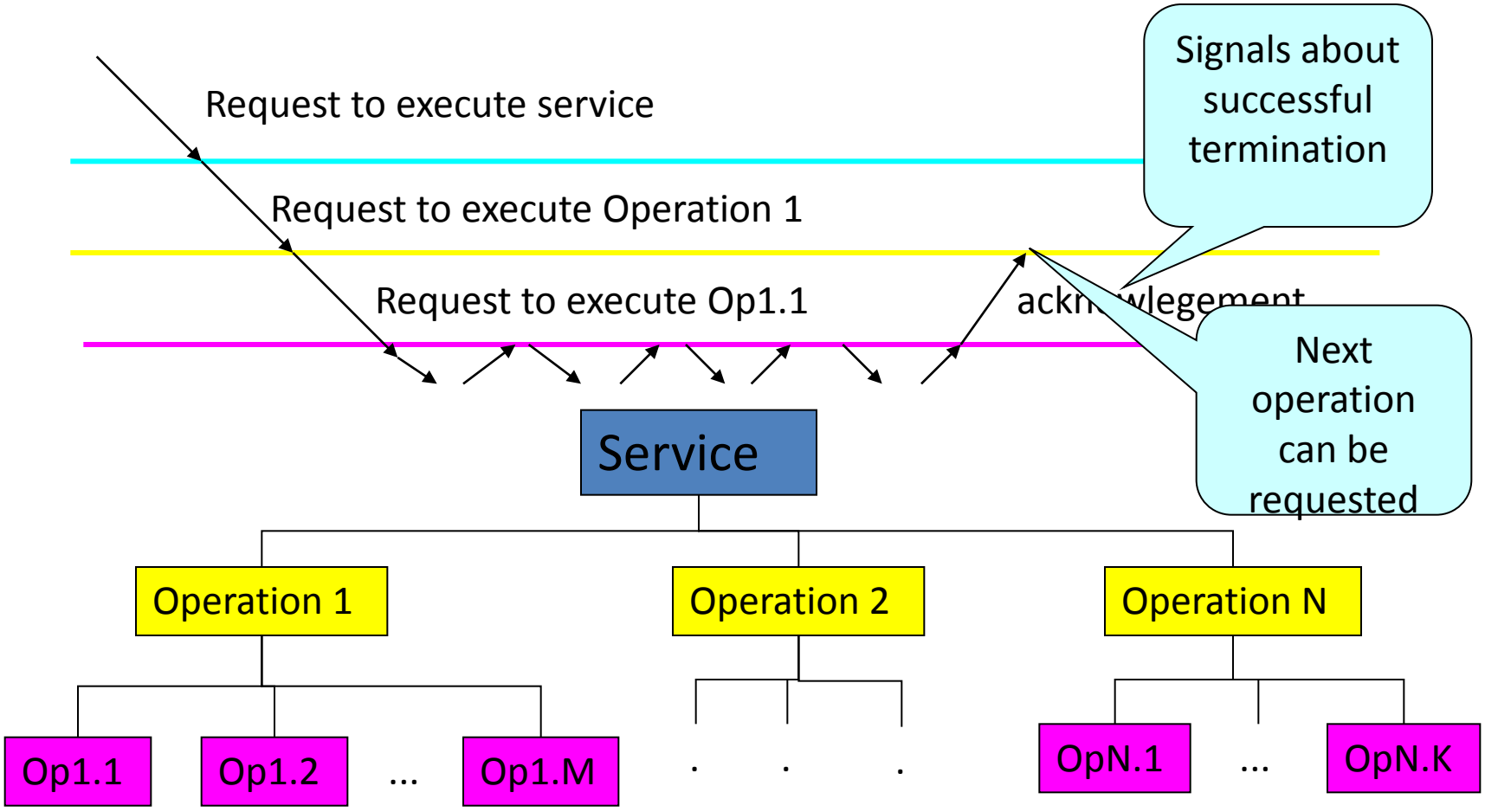
- Joint work with Perkin Elmer Life Science company (Finland)
- EU FP5 MATISSE project -- Methods and tools for industrial strength system engineering (2000-2003)
- Goal: ensure safety and reliability of a pharmaceutical robot

Control systems with layered architecture

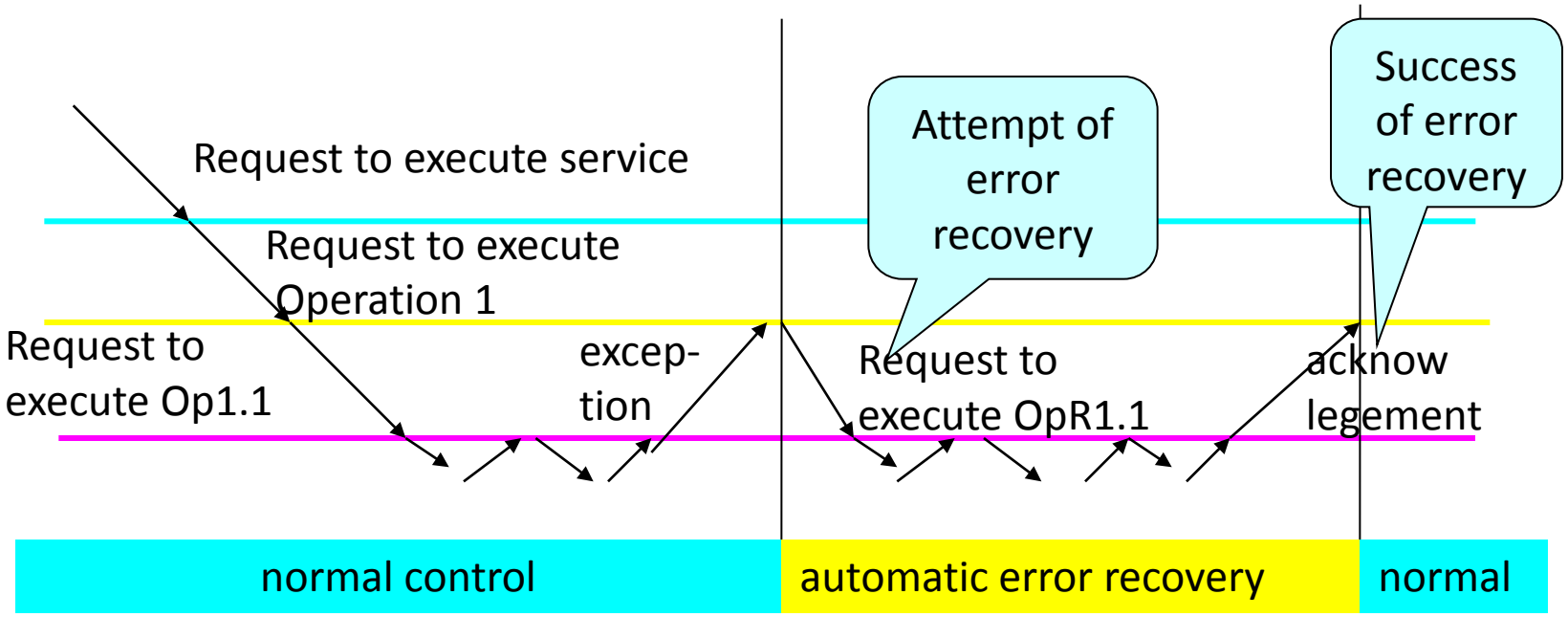
- The lowest layer: embedded subsystems that directly communicate with sensors and actuators
- The intermediate layers: components that encapsulate the lowest layers and provide interface to them
- The highest layer: a component server



Control systems with layered architecture



Exceptions in a layered architecture

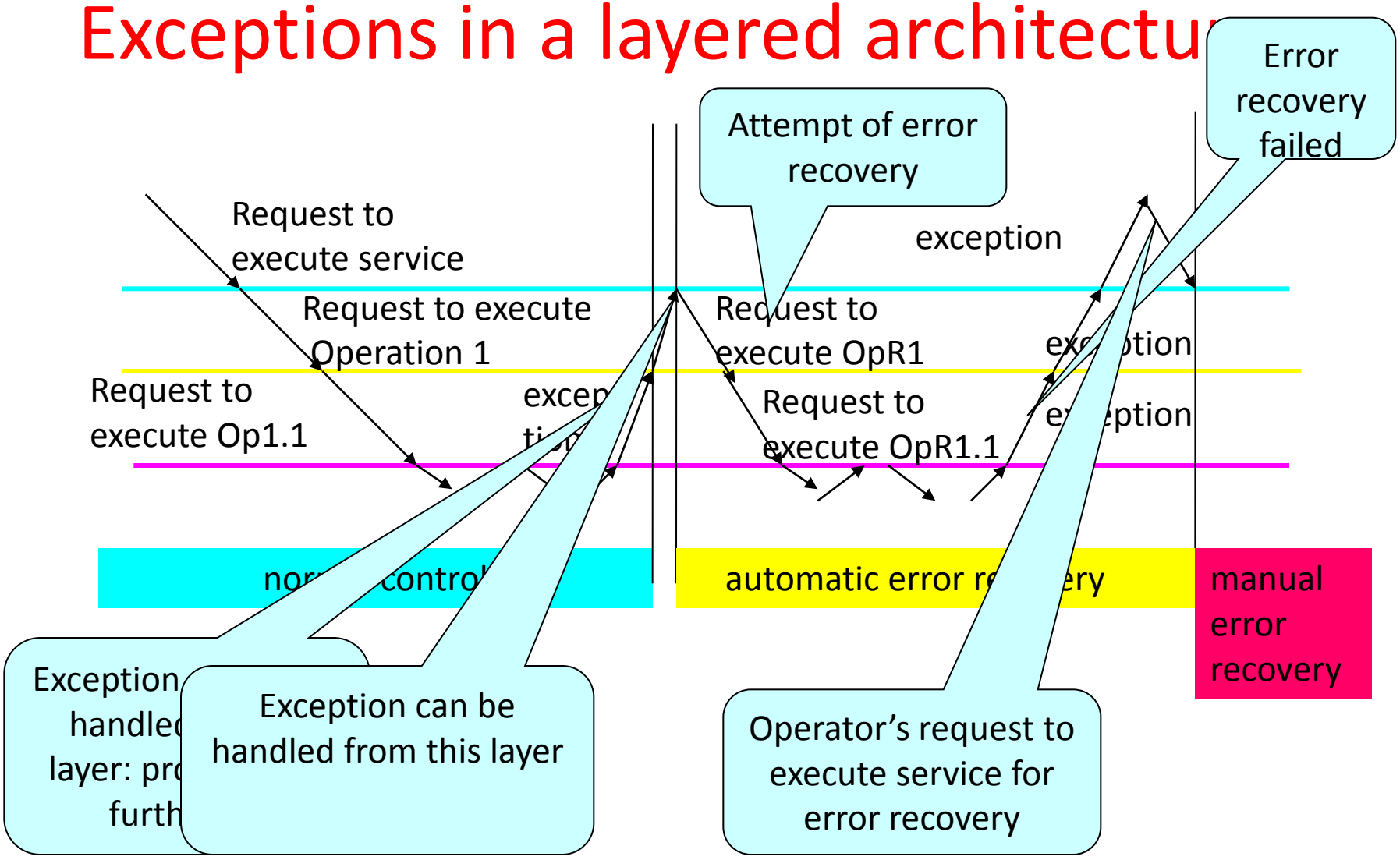


Exception signals about error occurrence

Error is manifestation of fault in a system component

Error recovery is an attempt to restore fault-free system state or at least to preclude system failure

Exceptions in a layered architectu



Exceptions

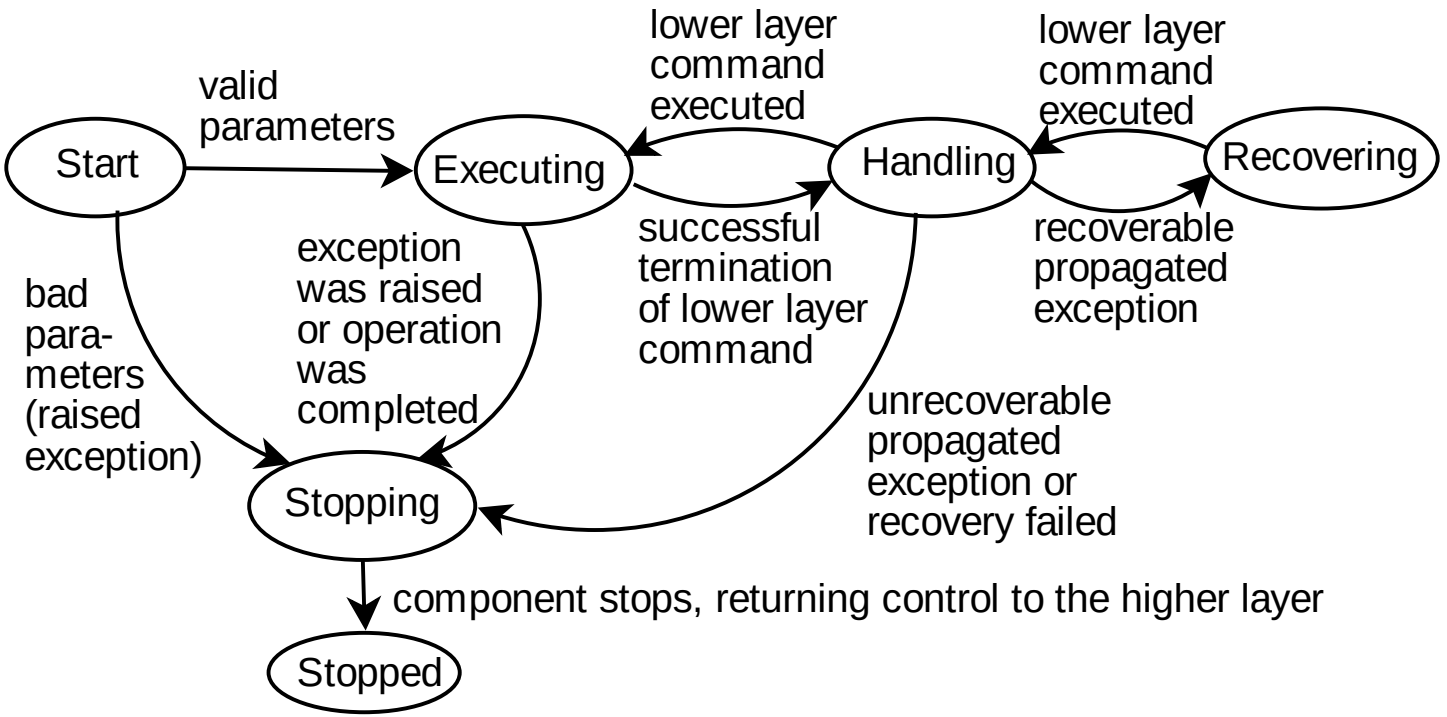
- For each component (except the lowest level subsystems) we can identify two classes of exceptions:
- 1. *generated exceptions*: the exceptions raised by the component itself upon detection of an error,
- 2. *propagated exceptions:* the exceptions raised at the lower layer but propagated to the component for handling.

Propagated exceptions

We classify them as

- an acknowledgement of normal termination, or
- a signal indicating recoverable error occurrence, or
- a signal indicating unrecoverable error occurrence.

Component's behaviour



Specification pattern of fault tolerant component

MACHINE

FTComponent

VARIABLES flag, …

INVARIANT flag :

{Executing,Handling,Recovering,Stopping,Stopped}

…

EVENTS

Start activates the component

Execute= WHEN flag=Executing

changes the current state; raises current layer exceptions; imitates execution of the lower layer

Specification of fault tolerant component

Handle = WHEN flag=Handling

evaluates lower layer exceptions. If they indicate success, enables Execute. If they are recoverable, enables Recover, otherwise Stop

Recover = WHENT flag=Recovering

models error recovery (by the lower layer command); passes control to Handle

Stop = WHEN flag=Stopping

terminates execution of component (if the current layer request is completed or the current layer exception is raised)

Discussion of abstract specification

From development perspective:

- Models the upper layer – the rest of the layers are “folded”
- Behaviour of the lower layer(s) is modelled by non-deterministic raising of its exception(s)

From specification perspective:

- Defines a general pattern for abstractly specifying a component at each layer

Refinement

- Each refinement step “unfolds” a lower layer
- The specification of the current layer is refined to
  - Add some implementation details, including activation of the lower layer component(s) in Execute and Recover operations
  - Block the current layer while the lower layer is executing a request

Refinement (cont.)

- The lower layer is specified according to the proposed specification pattern
- Refinement process continues until we reach the bottom layer

Discussion of the modelling approach

- A general formal specification pattern that can be recursively applied to specify fault tolerance mechanisms at each architectural layer
- Pattern can be iteratively applied via stepwise refinement in Event-B
- The approach results in development of a layered fault tolerant system correct by construction

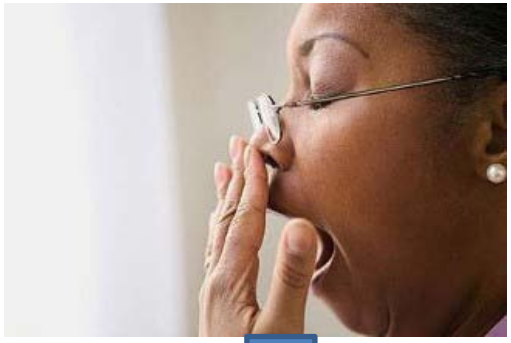
Mode-rich systems

- Modes – mutually exclusive sets of system behaviour (Leveson) -- are widely used in industry

Motivation

Desirable mode transition

ActiveAvarenessMode



TiredMode

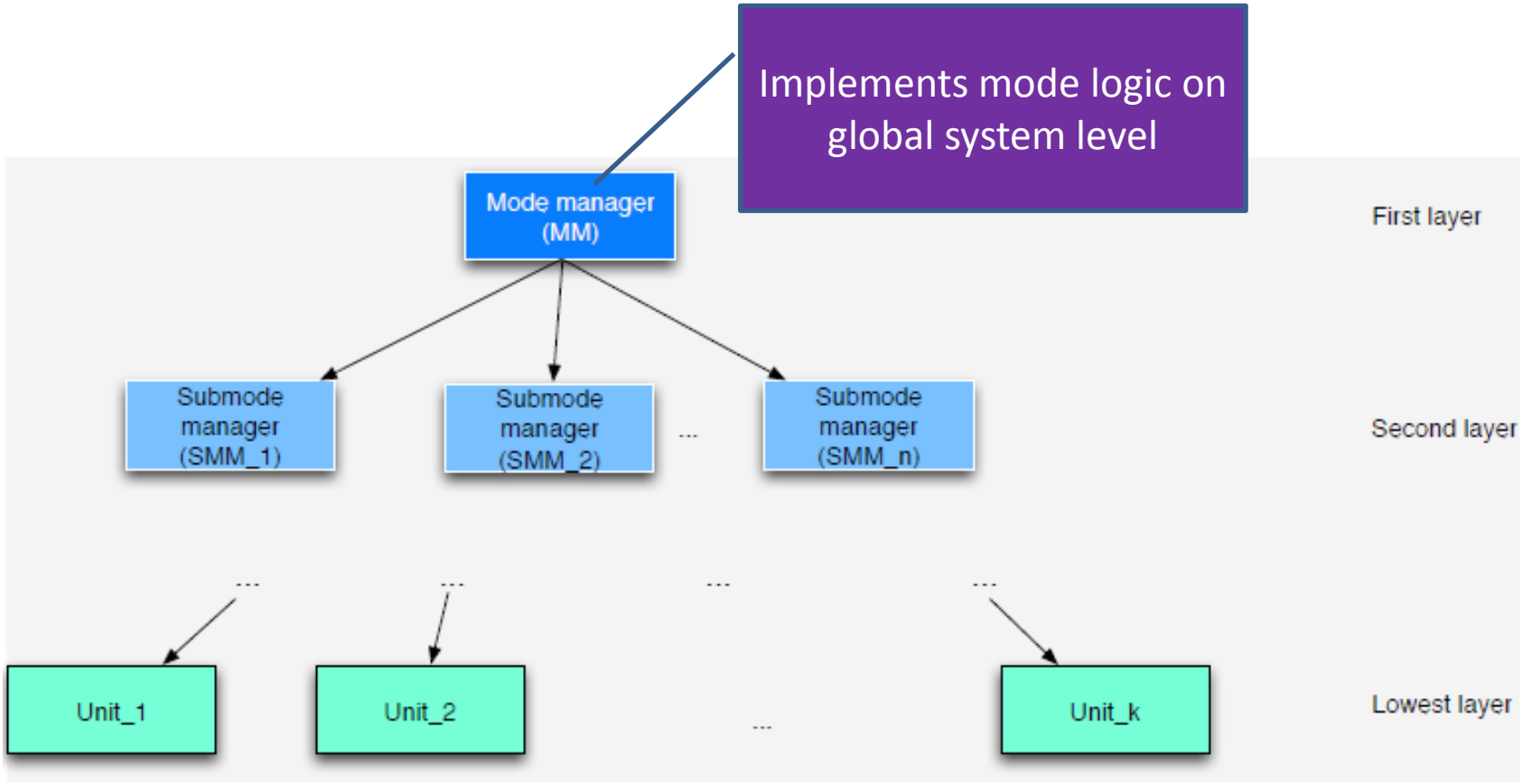
SleepingMode

Undesirable mode transition

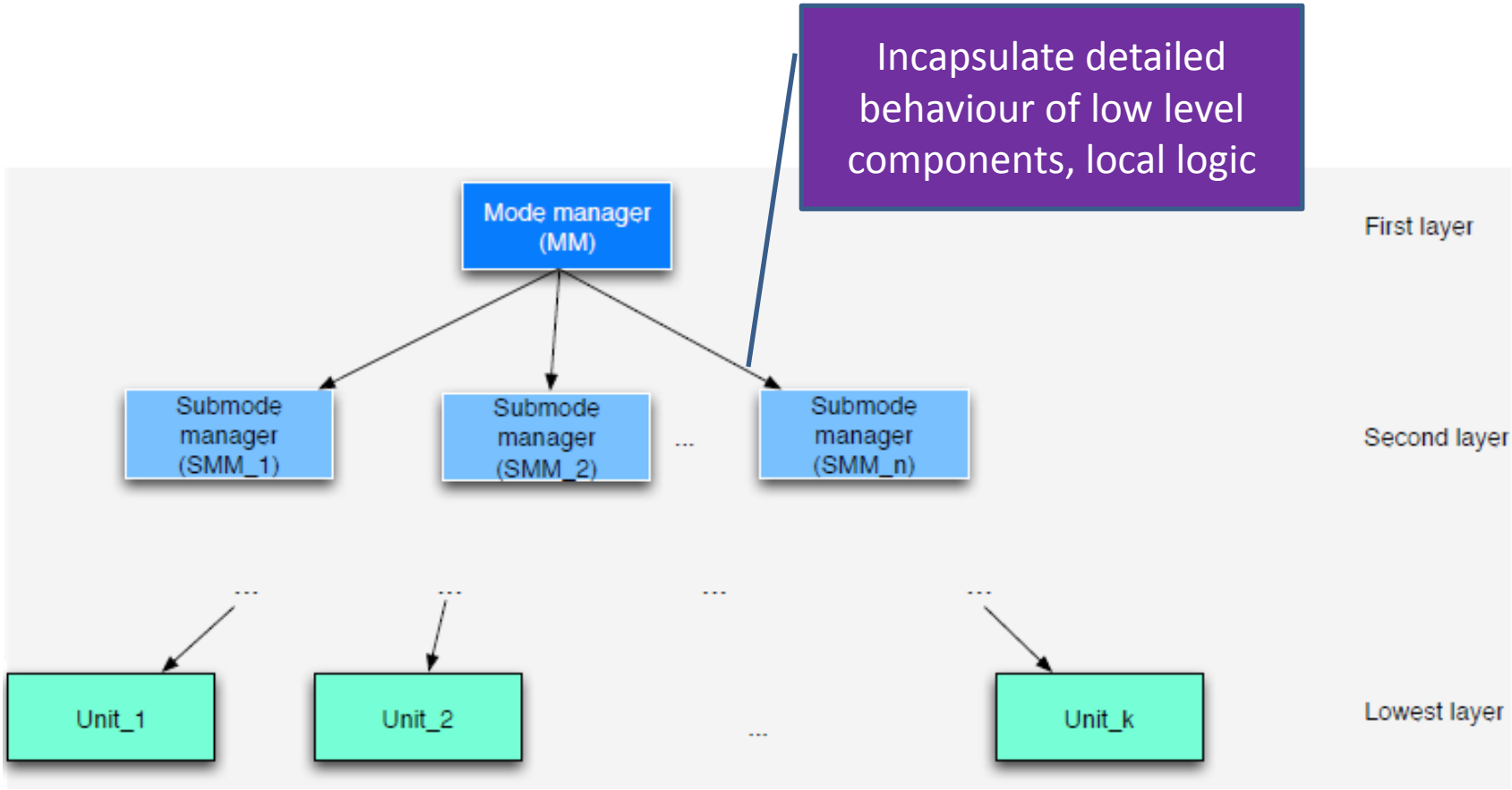
Motivation

- Modes – mutually exclusive sets of system behaviour (Leveson) -- are widely used in industry
- Complex mode transition scheme:
  - Long-running mode transitions of components
  - Strong impact of component failures on mode transition scheme
- Lack of generic archiectural-level approaches faciliatating design and verification of mode-rich systems

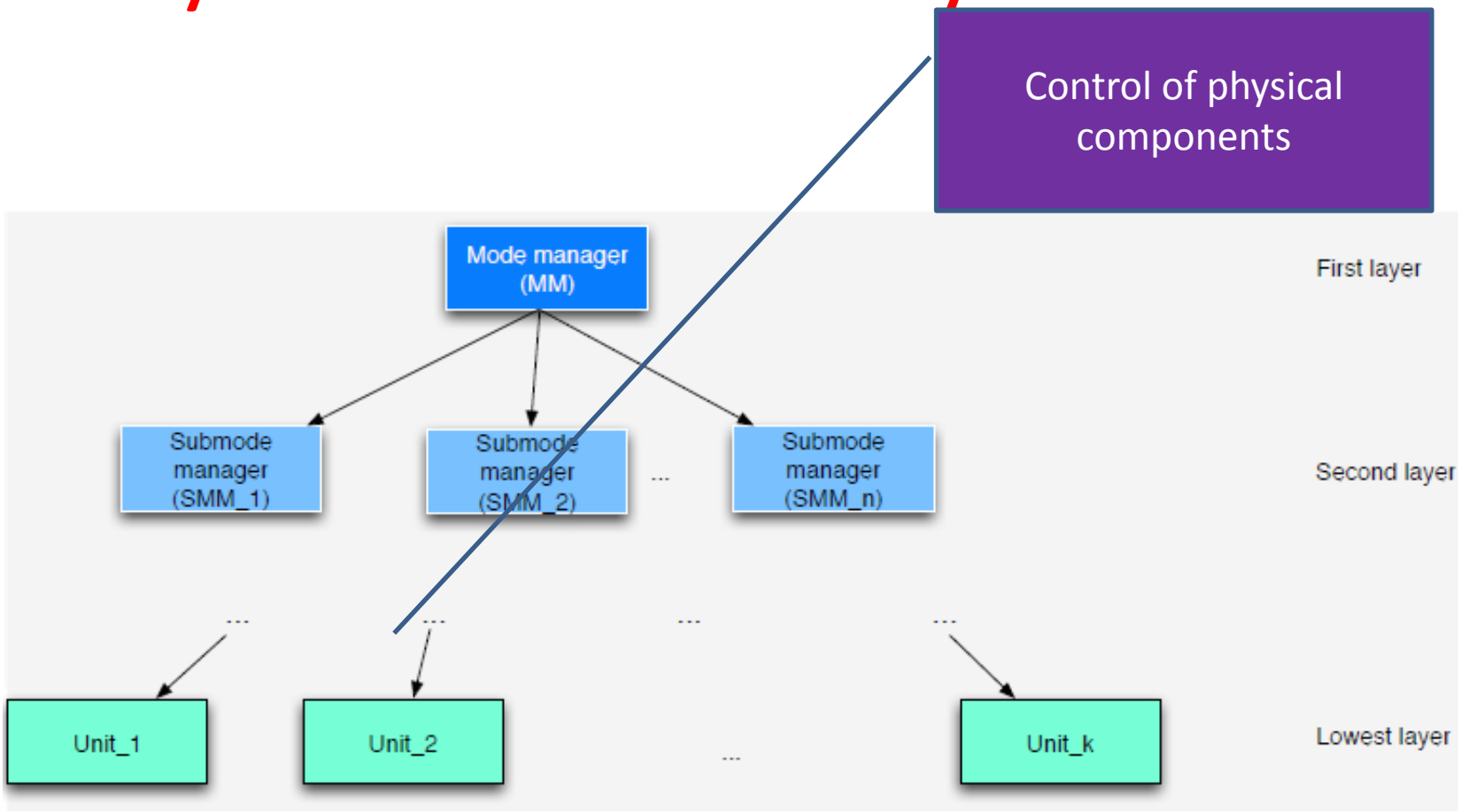
Layered mode-rich systems



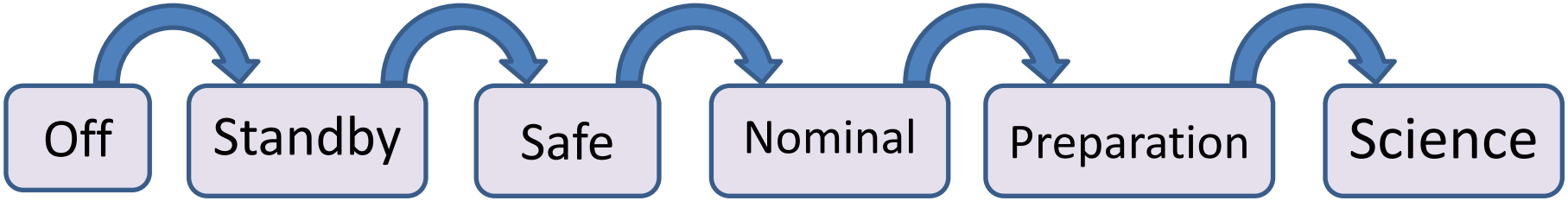
Layered mode-rich systems



Layered mode-rich systems



Attitude and Orbit Control System (AOCS): Global mode logic

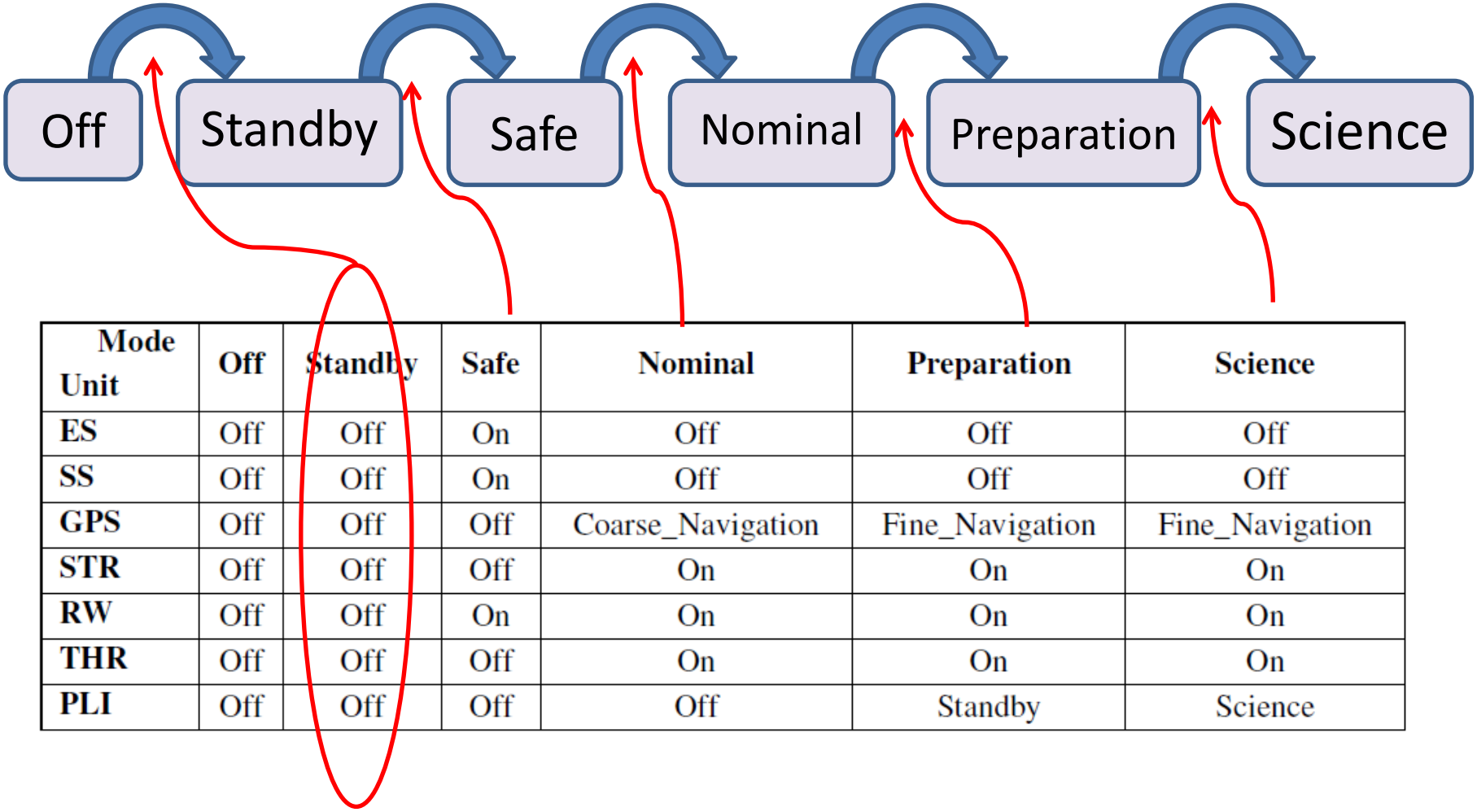


- Off– the satellite is in this mode after system (re)-booting
- Standby mode is maintained until separation from the launcher is completed
- Safe – The satellite acquire stable attitude, which allows the coarse pointing control
- Nominal -- The satellite is trying to reach the fine pointing control, which is needed to use the payload instruments
- Preparation – The payload instrument is getting ready after fine pointing is reached
- Science – the payload instrument is ready to perform its tasks. The mission goal is to reach this mode and stay in it as long as needed

AOCS Components

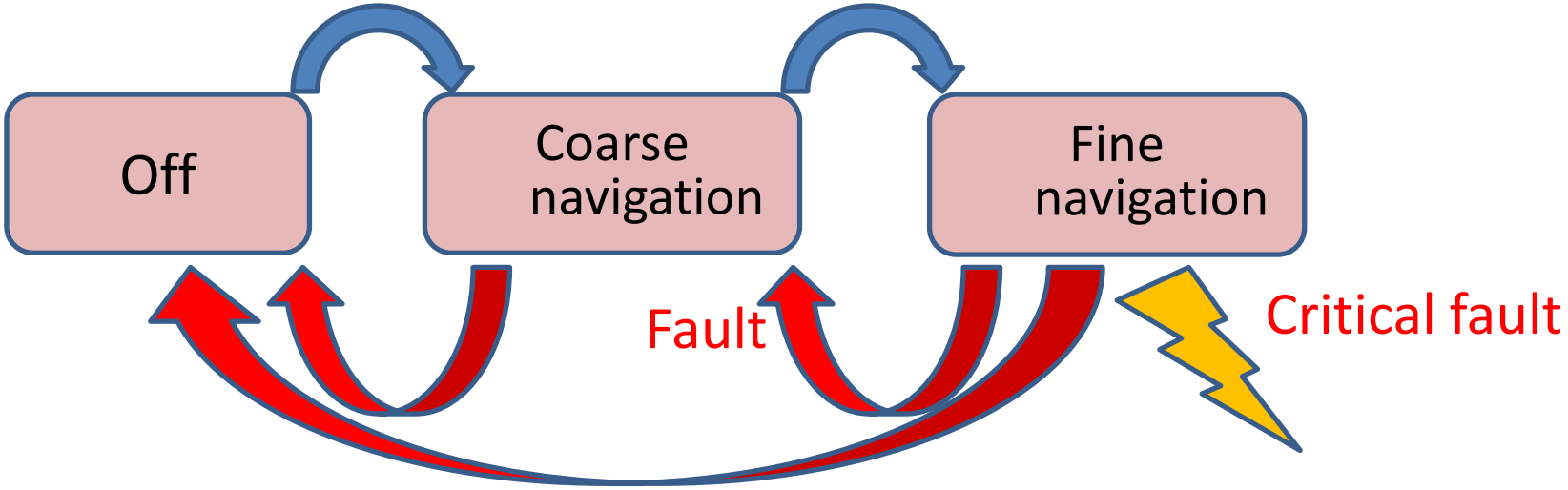
- Four sensors
  - Star tracker, Sun Sensor, Earth Sensor, Global Positioning system
- Two actuators
  - Reaction Wheel and Thruster
- Payload instrument producing mission measurements

AOCS: Mode entrance conditions



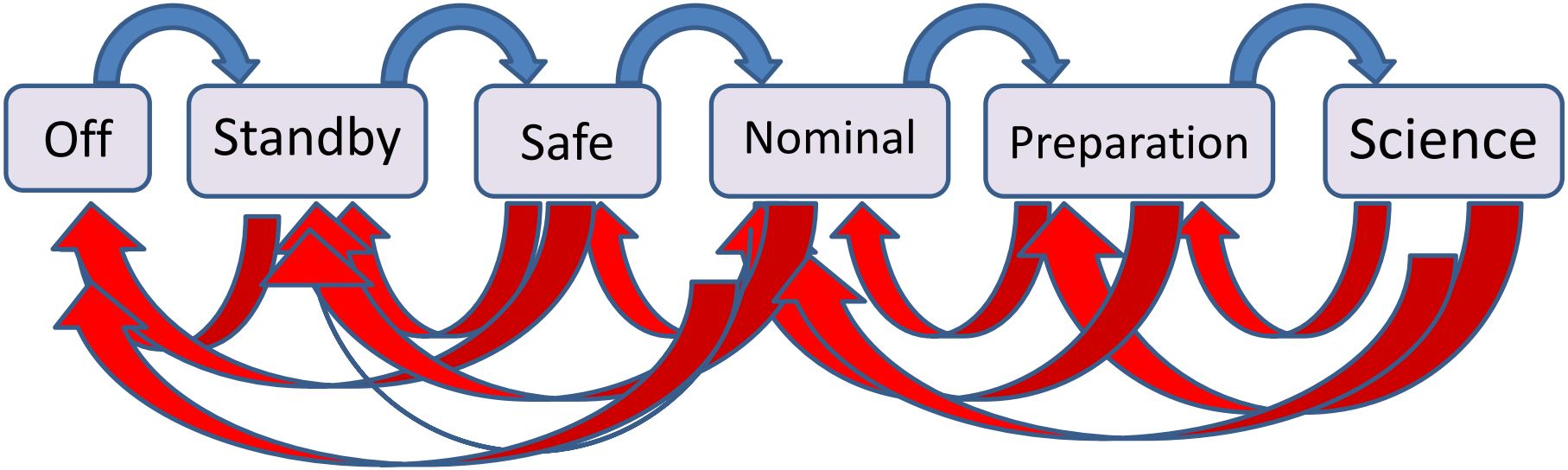
Mode / Unit	Off	Standby	Safe	Nominal	Preparation	Science
ES	Off	Off	On	Off	Off	Off
SS	Off	Off	On	Off	Off	Off
GPS	Off	Off	Off	Coarse_Navigation	Fine_Navigation	Fine_Navigation
STR	Off	Off	Off	On	On	On
RW	Off	Off	On	On	On	On
THR	Off	Off	Off	On	On	On
PLI	Off	Off	Off	Off	Standby	Science

Fault occurrence and mode logic



- While trying to reach a certain mode a component can fail and roll-back
- In some cases the entire system needs to roll-back

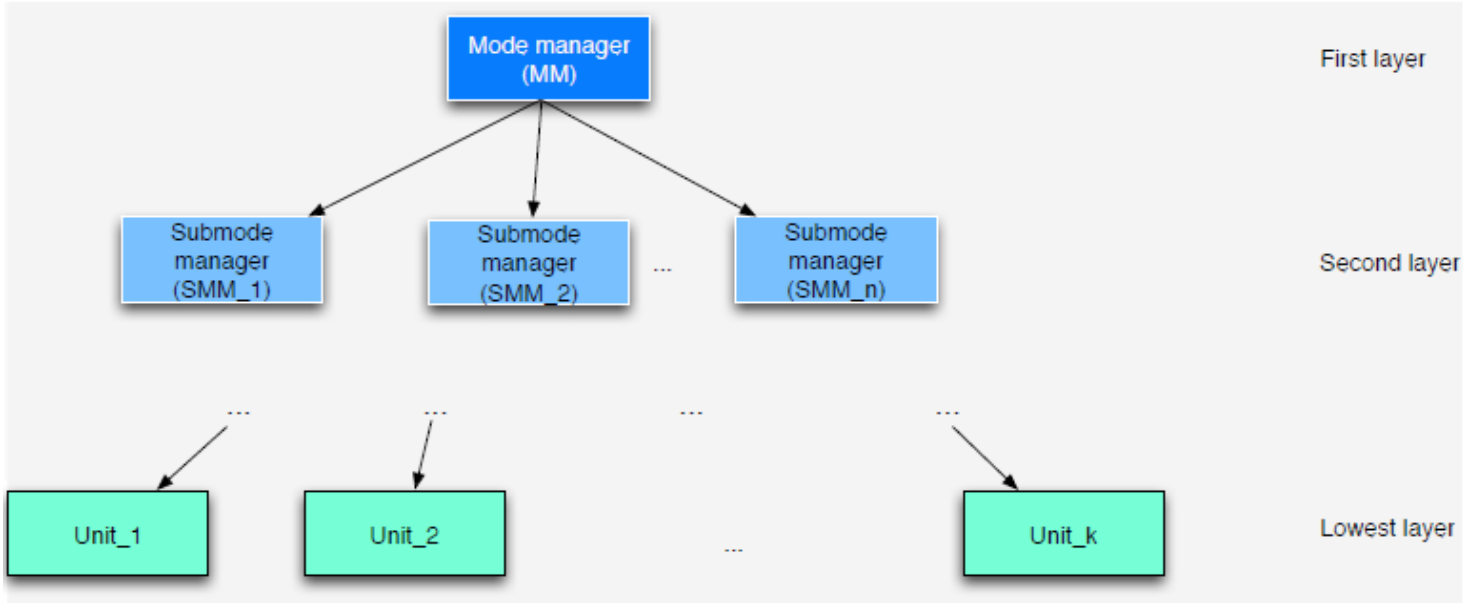
Attitude and Orbit Control System (AOCS): Global mode logic



- Several component might fail at the same time or during roll-back
- Cascading effect
- State explosion problem, very large number of scenarious and hence difficult to test

We need an architectural-level rigorous approach to designing mode-rich systems to handle complexity and guarantee correctness

System structure and behavior



The system is cyclic.

System structure and behavior

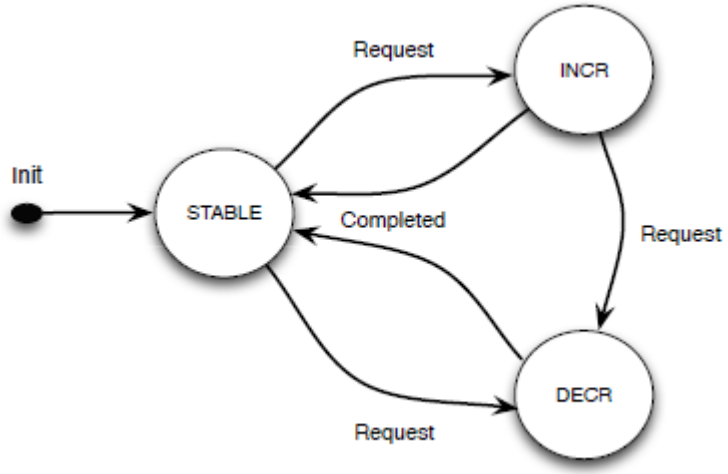
At each cycle MM assesses SMM states by monitoring their modes and detected errors and either

- Initiates a forward transition according to the predefined scenario
- Initiatates backward transition (if error occurred). Target mode depends on error severity
- Completes transition to the target mode and becomes stable (if cond. for entering mode are satisfied and no error occur)
- Maintains the current mode (if neither cond. for entering new mode are satisfied nor error occured)

Mode managing component: behavioural pattern

Introducing component status:

- *last_mode* – last successfully reached mode
- *next_target* – the target mode that a compoent is currently in transition to
- *previous_target* – the previos mode that a component was in transition too (though not necessarily reached it)



$$Stable \triangleq last\_mode = previous\_target \wedge next\_target = previous\_target$$

$$Increasing \triangleq last\_mode = previous\_target \wedge previous\_target < next\_target$$

$$Decreasing \triangleq next\_target < previous\_target$$

Mode managing component:

- Stable state: decide to initiate a new mode transition to some more advanced mode
- Transition state: monitor states of lower layer components. If at some point mode entry conditions are satisfied for the target mode then complete transition and become stabe
- In both stable and transitional states: monitor lower layer components for the detected error, execute error recovery by setting new target mode if errors are detected

Refinement and modularization

- Modularization: we model a component via its interface and develop its implementation as separate (formal) development without losing correctness

Formal development strategy

- General idea: to define generic interface of mode managing component
- Build the entire system in the top-down fashion by instantiating generic interface and unfolding one layer at the time
- Proof desired properties of model logic as part of refinement verification

Generic interface of mode-managing component

INTERFACE *MMC_I*
SEES *MMC_Context* (* introduces abstract *Modes*, *Errors*, and *Next* *)
VARIABLES *last_mode*, *next_target*, *previous_target*, *error* (* list of external variables of a module *)
INVARIANT
types of external variables
other invariant properties (* mode status properties *)
OPERATIONS

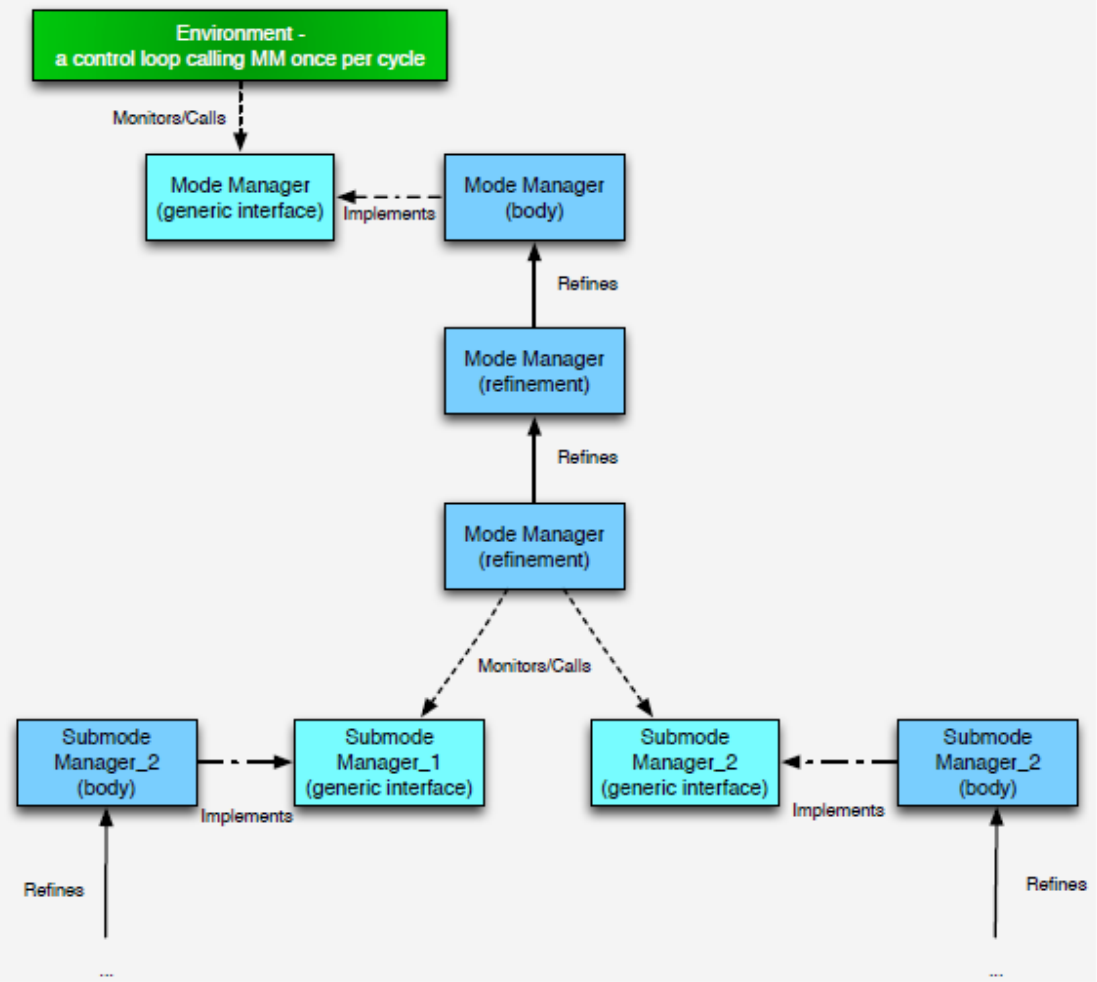
SetTargetMode =
ANY *m*
PRE
Component has not failed
m is a new target mode
POST
new target mode is set

ResetError =
PRE
the error flag is raised
POST
the error flag is cleared

RunStable =
PRE
Component is stable and not failed
POST
Component either remains stable
or changes its mode according to the scenario
or raises the error flag

RunNotStable =
PRE
Component is in a mode transition
POST
A mode transition is completed
or a mode transition continues
or the error flag is raised

Refinement strategy



Proved properties of mode logic

- Unambiguity of mode logic:

$$\forall i, j \bullet M_i \in Modes \ \wedge \ M_j \in Modes \ \wedge \ i \neq j \ \Rightarrow$$
$$Mode\_ent\_cond(M_i) \cap Mode\_ent\_cond(M_j) = \varnothing$$

- A component satisfies mode entry conditions and mode invariant (when it is *Stable)*

$$\forall i \bullet m_i \in Modes \wedge current\_mode = M_i \wedge Stable \ \Rightarrow \ Mode\_ent\_cond(M_i)$$
$$\forall i \bullet m_i \in Modes \wedge current\_mode = M_i \wedge Stable \ \Rightarrow \ Mode\_Inv(M_i)$$

Service-oriented systems

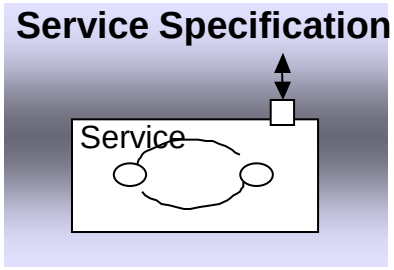
- Telecommunication systems:
  - Distributed software-intensive systems
  - Provide a large variety of services

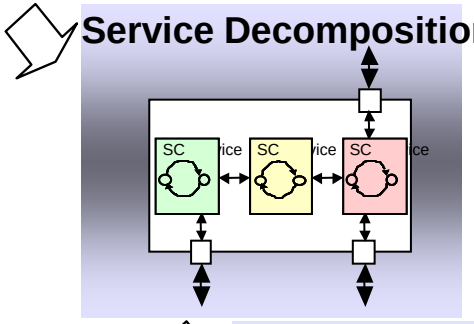
Important to guarantee correctness of software and system fault tolerance

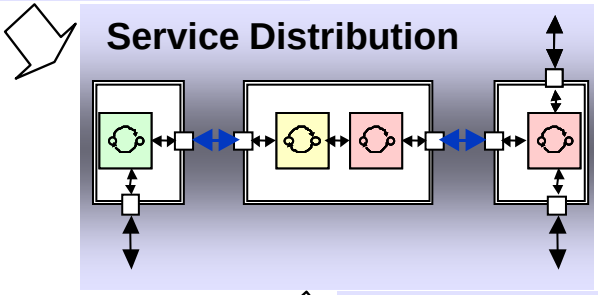
The Lyra Design Method

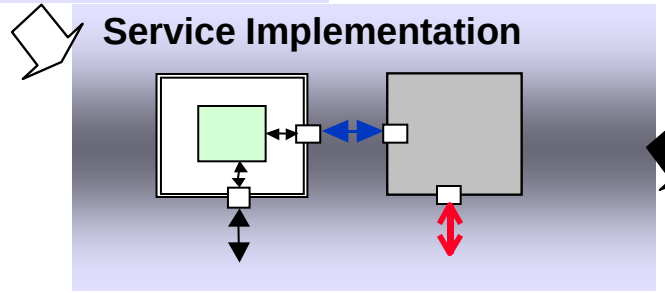
- UML2-based service-oriented method for developing communicating systems
- The system behaviour is modularised and organised into layers according to external communication interfaces
- Distributed network architecture is derived via number of model transformations

Lyra Design Method

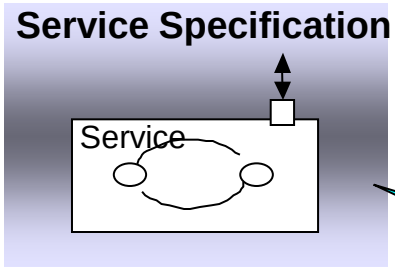


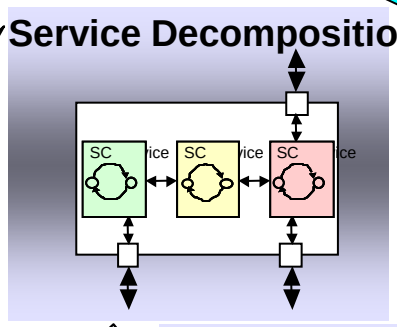


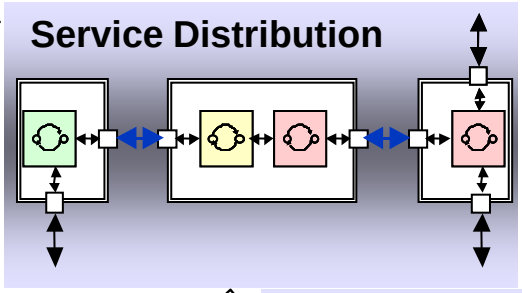


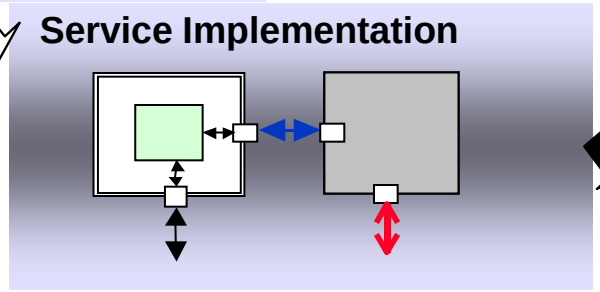


Lyra Design Method



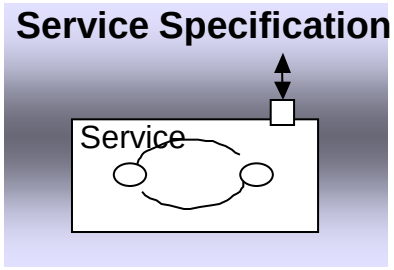


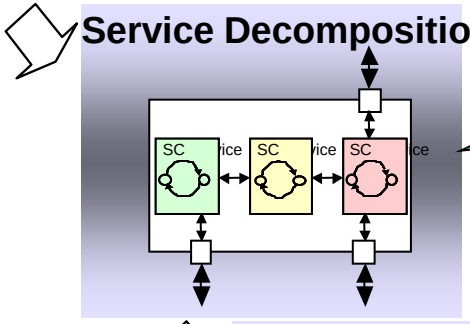




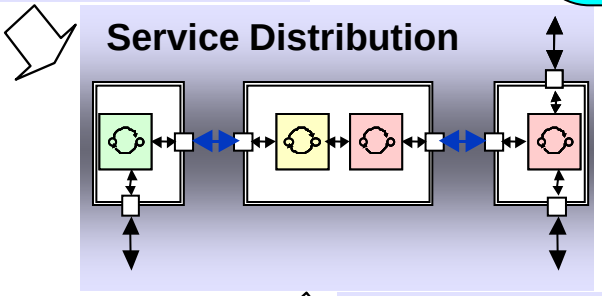
Service specification: system-level services and interfaces are defined

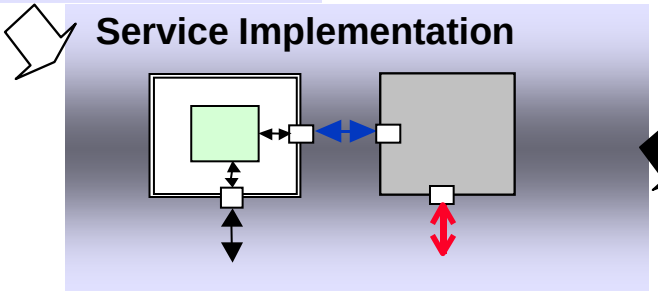
Lyra Design Method



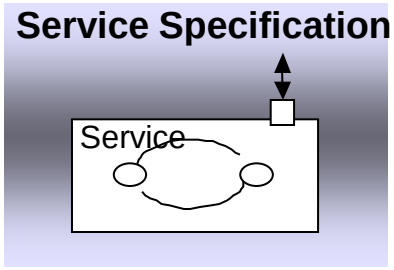


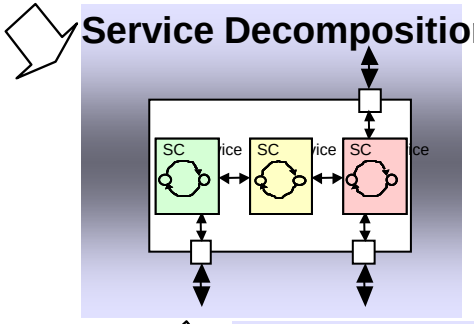
Service decomposition:the abstract model is decomposed into a set of service components and interfaces between them

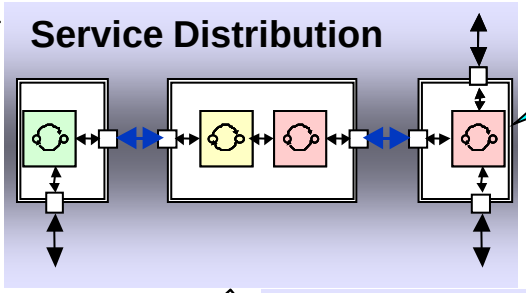




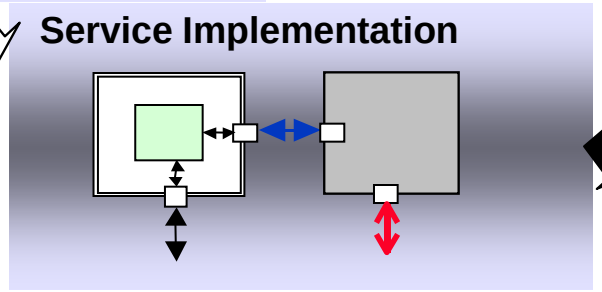
Lyra Design Method



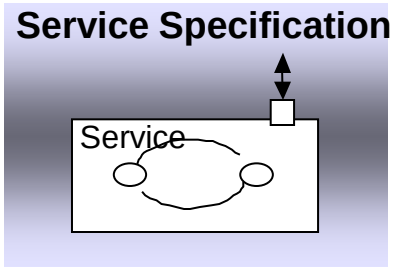


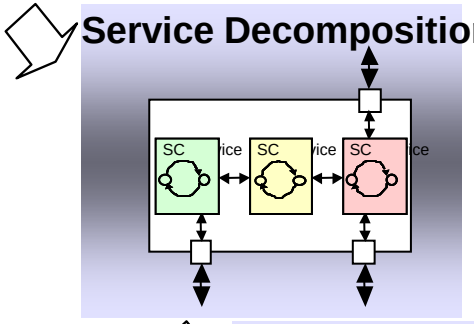


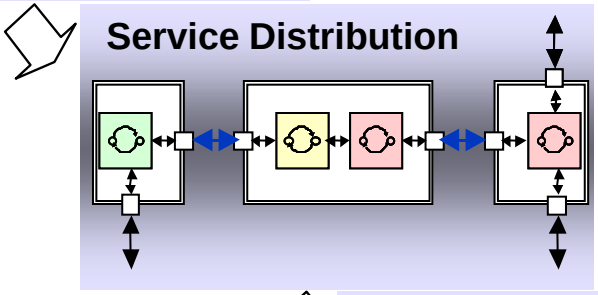
Service distribution: the logical architecture of services is distributed over a given network

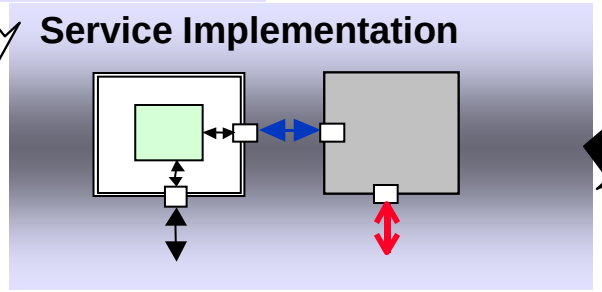


Lyra Design Method

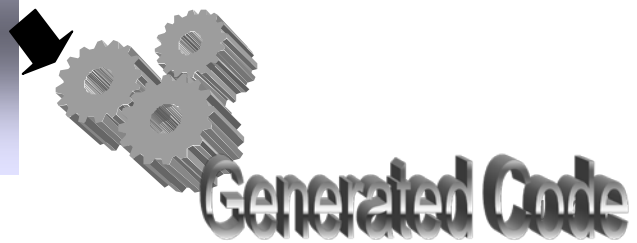








Service implementation: low-level implementation details are added and platform specific code is generated



Formal Development

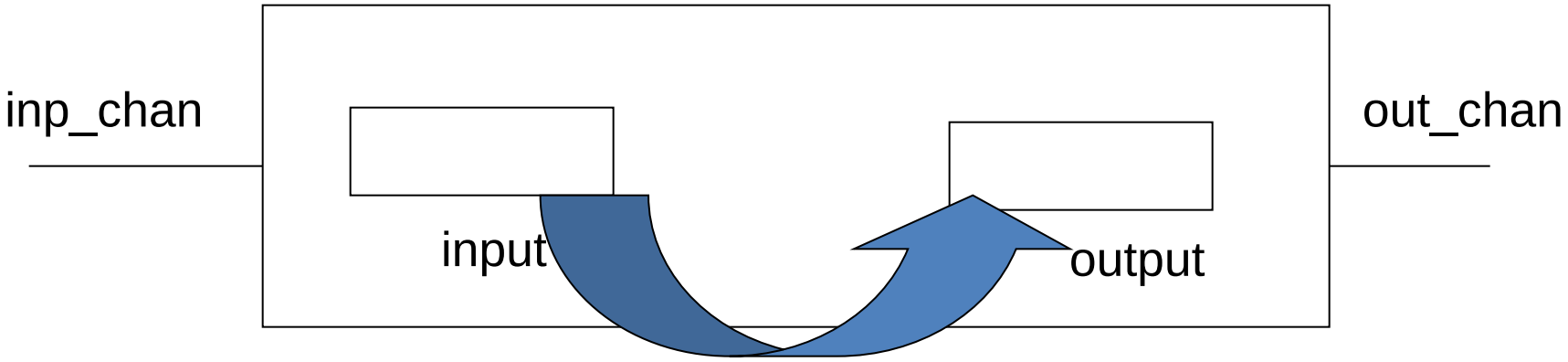
- We single out a generic concept of a communicating service component and propose patterns for specifying and refining it

- In the refinement process a service component is decomposed into service components of smaller grannularity according to the same pattern

Modelling a Service Component in B

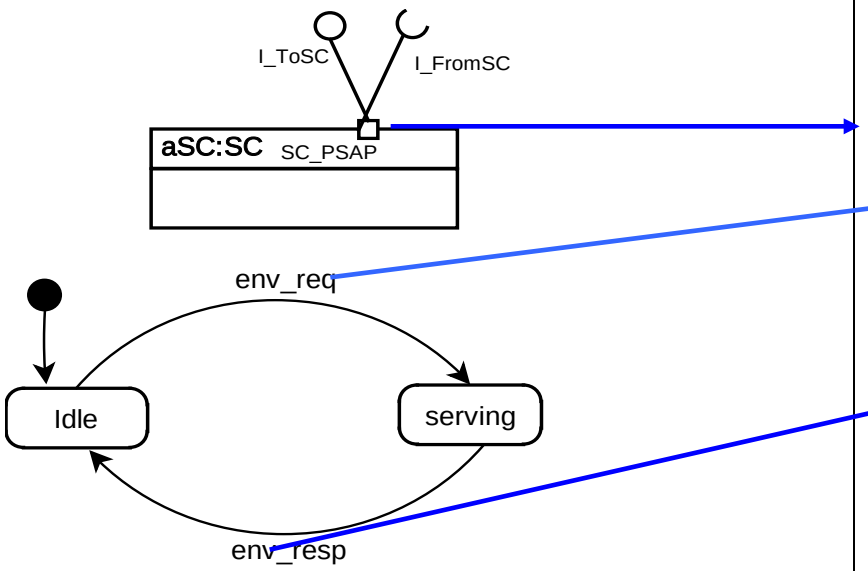
- Components are created according to pattern ACC - Abstract Communicating Component
- ACC Component consists of
  - a "kernel", i.e., the provided functionality
  - "communication wrapper", i.e., the communication channels via which data are supplied to and consumed from the component

Behaviour of Abstract Communicating Component



calculate

Translating UML2 model into the ACC pattern

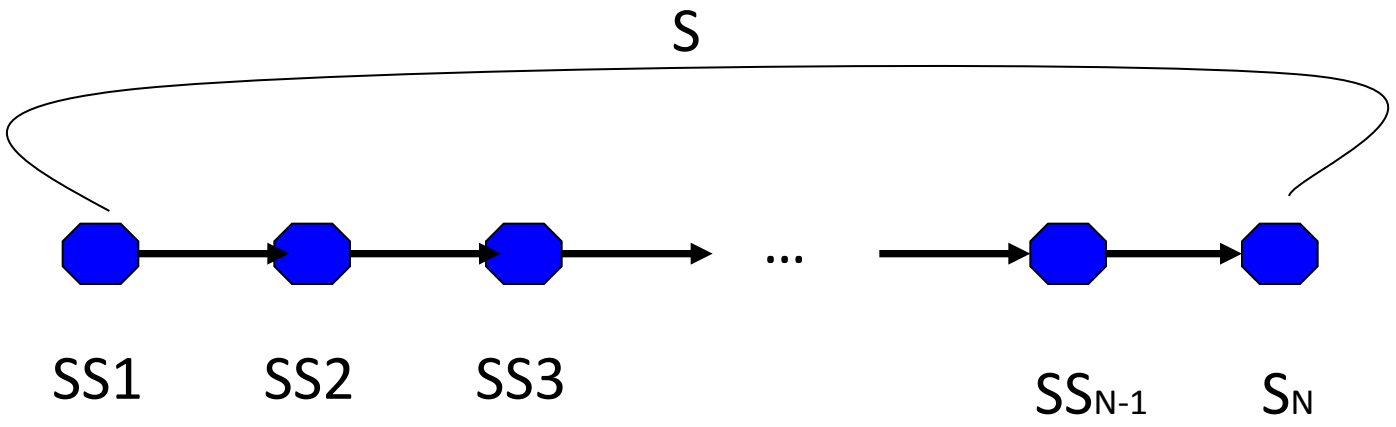



```
MACHINE ACC
....
EVENTS
/* communicational */
    env_req
    read
    write
    env_resp

/* functional */
    calculate

END
```

Service Decomposition Phase

- External service providers are introduced
- The behaviour is decomposed accordingly

Service decomposition: fault free execution flow

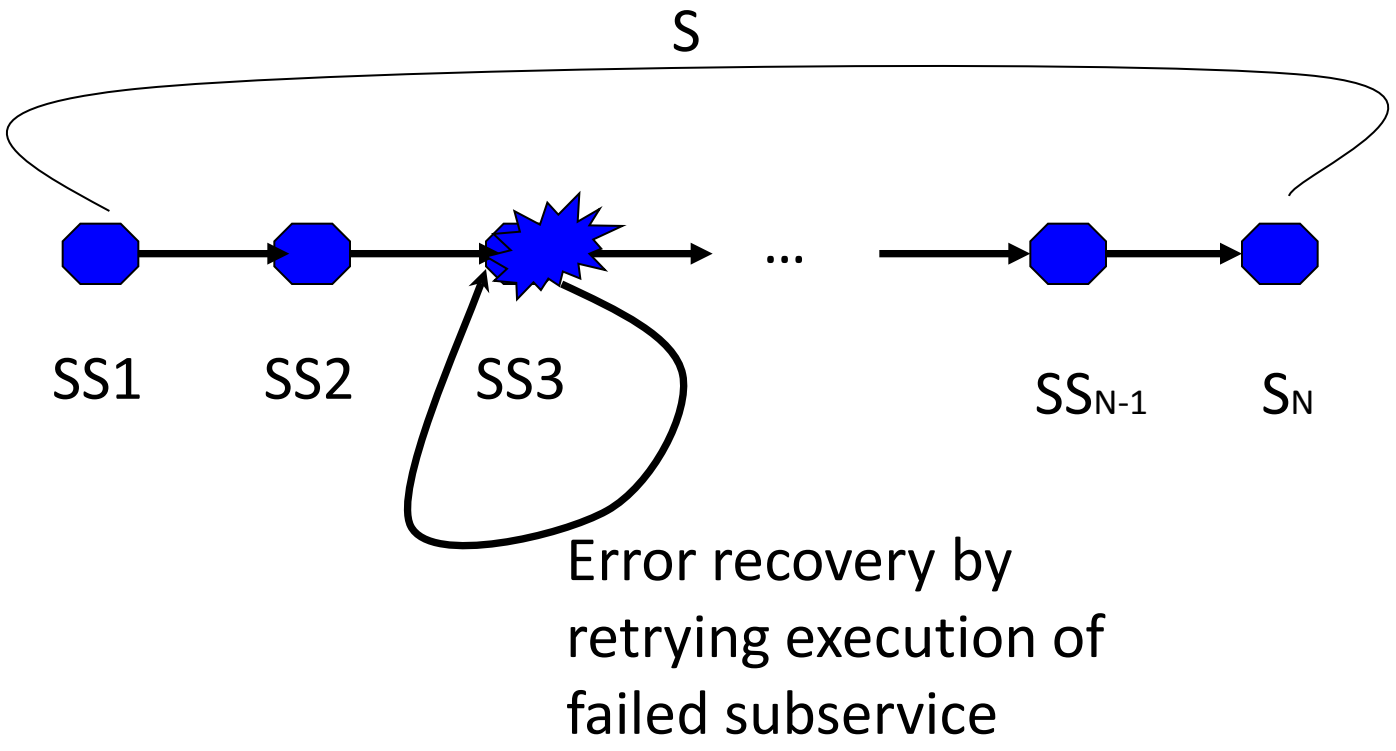


Fault Tolerance

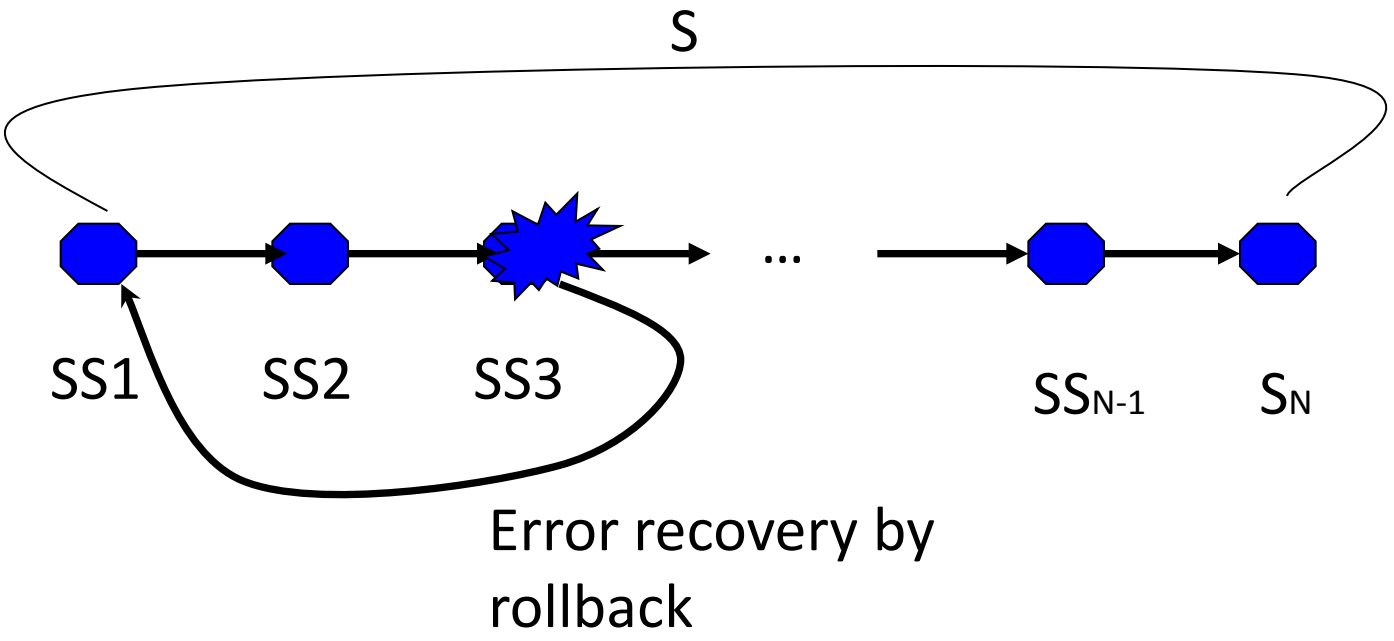
- Initial stage: not only successful but also failed service provision

- Decomposition: each subservice can fail

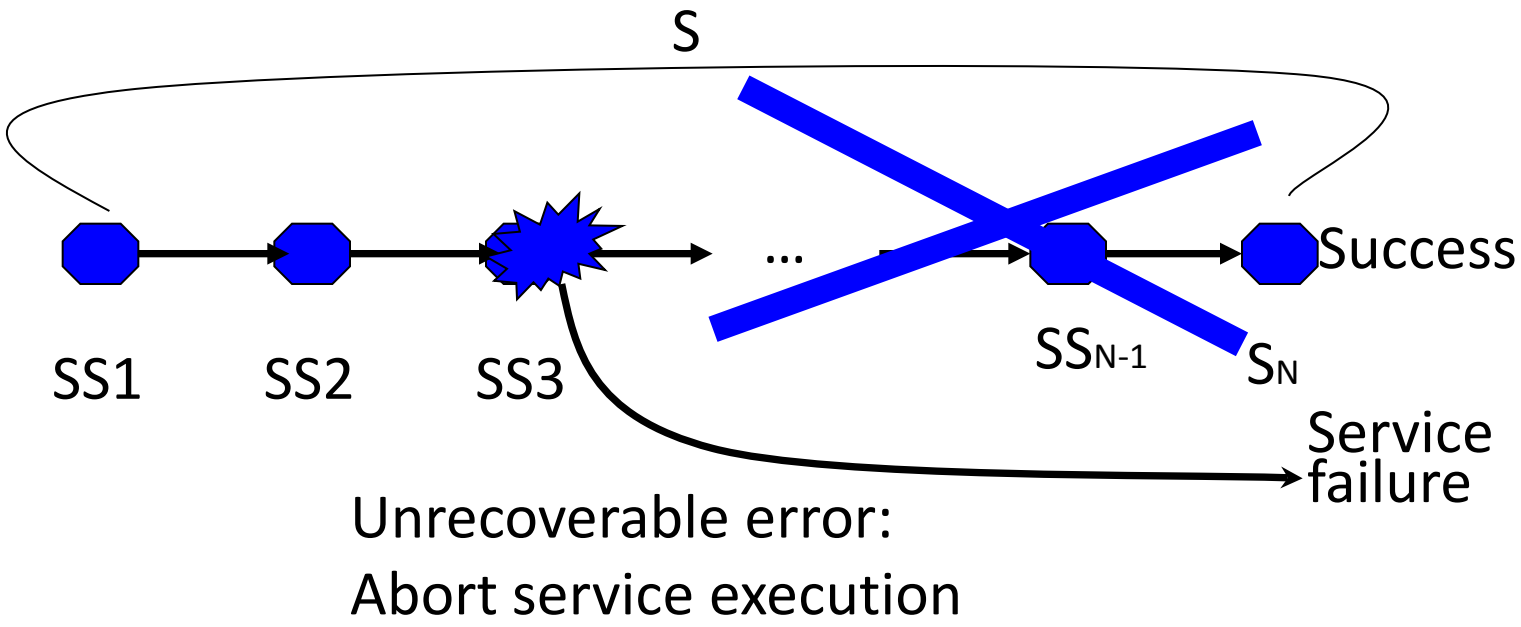
Service decomposition: faults in execution flow



Service decomposition: faults in execution flow

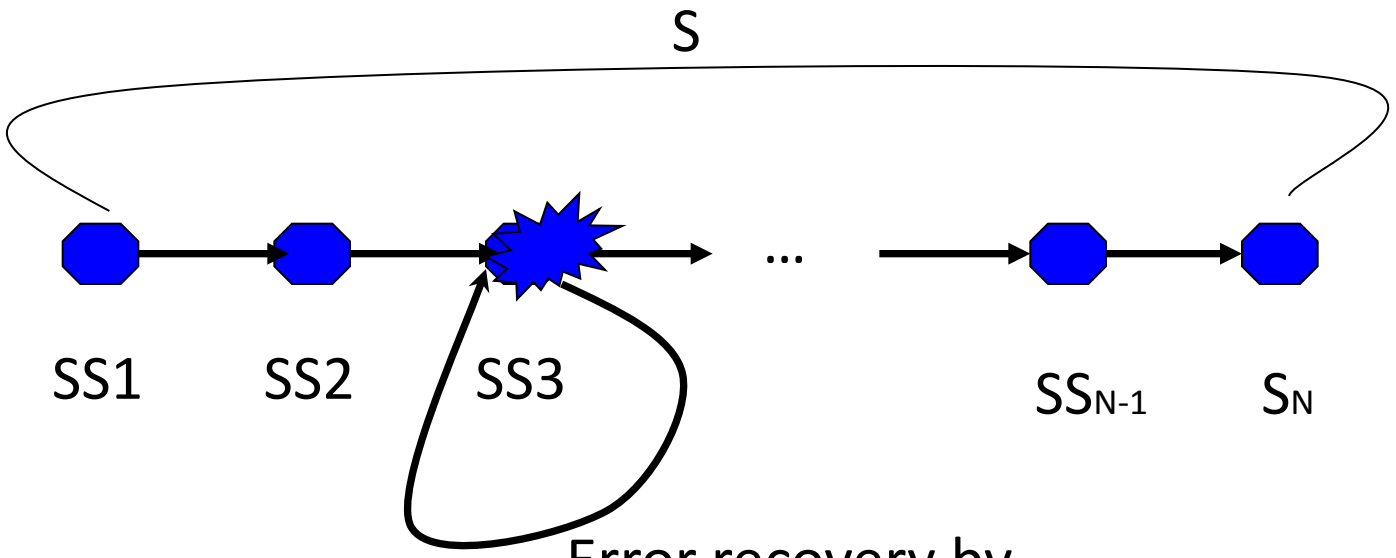


Service decomposition: faults in execution flow

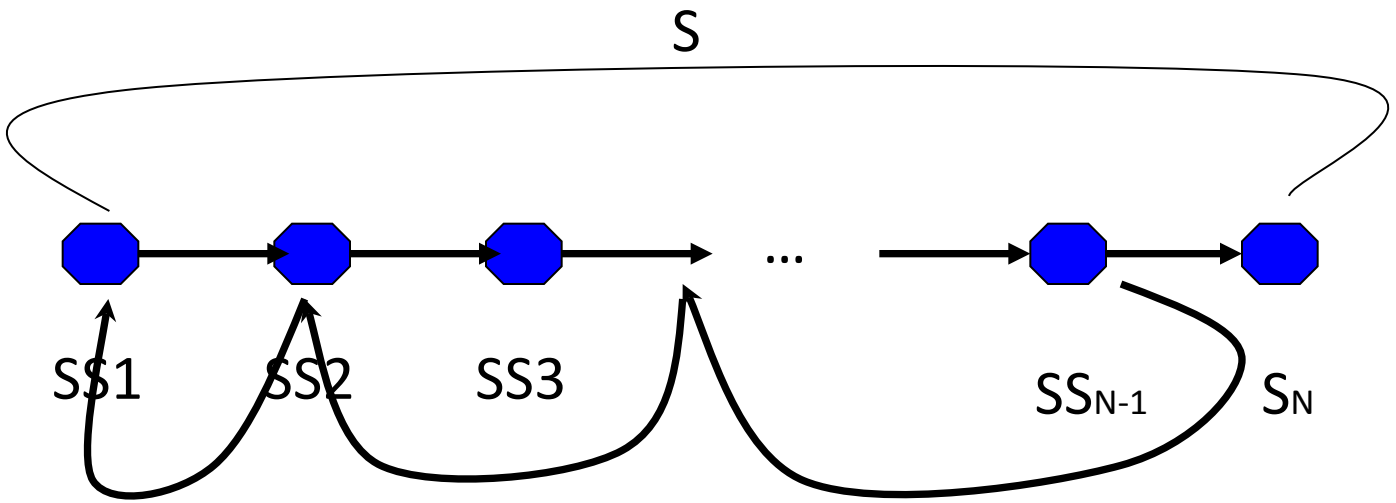


Unrecoverable error:
Abort service execution

Convergence of error recovery?



Convergence of error recovery?

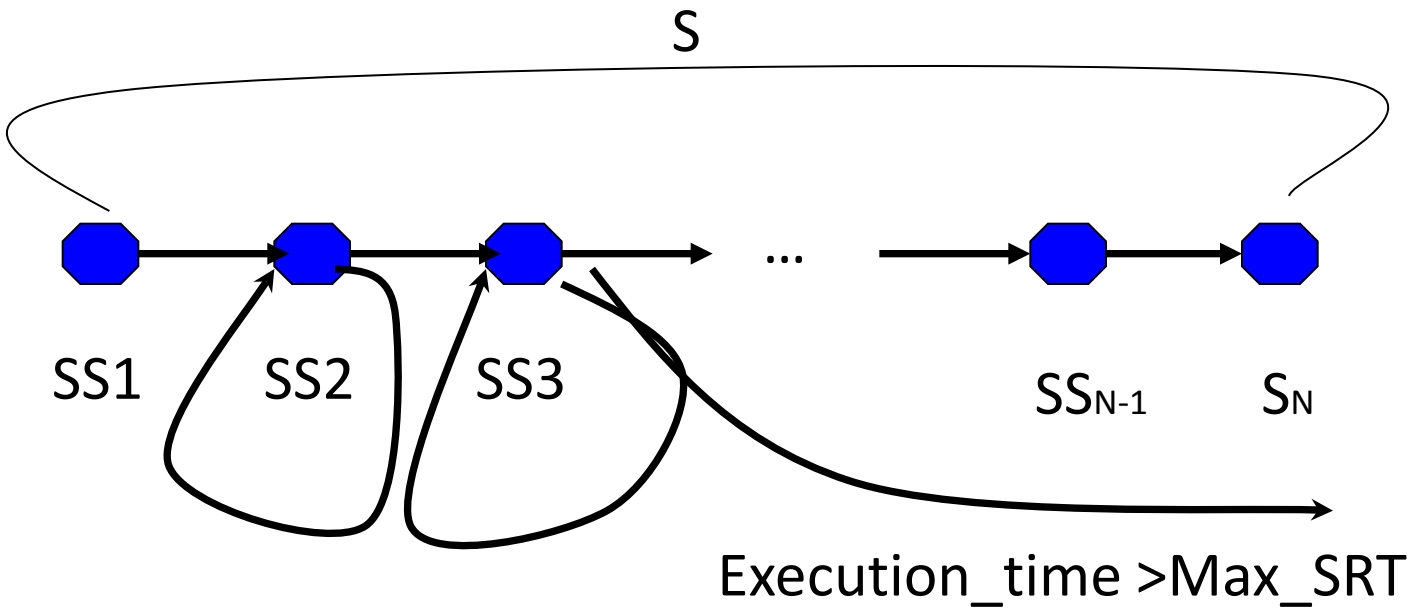


Error recovery by
rollback: domino effect

Convergence of error recovery

- How to bound error recovery?
- We introduce Maximal Service Response Time (Max_SRT)
- If the service fails to complete computation within Max_SRT then it aborts and returns failure response

Abort of service due to timeout



Service Decomposition Phase

- In B model: decomposition is represented as refinement of the initial abstract pattern ACC
- B refinement step focuses on the ”functional” part of the specification
- We introduce the operation Service_Director and Time
- Service_Director orchistrates execution flow
- Time non-deterministically decrements the execution time left

Service Distribution Phase

- This phase describes how service components are distributed over a given network
- Service Distribution phase of Lyra corresponds to one or several B refinements
- Refinement steps introduce separate B components modelling external service components
- All new B components are specified according to the same (ACC) pattern

Probabilistic extension

- Goal: to integrate quantitative reasoning about dependability in Event-B refinement
- Extending the language and semantics of Event-B to enable dependability analysis using the theory of Markov processes
- Qualitative probabilistic choice x $|\oplus$ P(v; x0)
  - Assigns x a new value x0 with some fixed (but unknown) probability
  - allows the reasoning about the fairness
  - can be placed only instead of an existing nondeterministic assignment (S. Hallerstede and T. S. Hoang. 2007)

Explicit probabilistic assignment

- Quantitative probabilistic assignment – discrete time

  $x \mid\!\oplus\ x_1\ @P_1;\ \ldots;\ x_m\ @P_m;$

  - assigns to x a new value x_i with some fixed and known non-zero probability p_i
  - defines a next-state distribution for any state in which event is enable
  - always refines its corresponding nondeterministic counterpart

Explicit probabilistic assignment

- Quantitative probabilistic assignment – continuous time

  $x \mid \oplus\ x_1\ @\lambda_1;\ \ldots;\ x_m\ @\lambda_m;$

  here λ_1 rate

  - With probabilities: Markov decision process (MDP) or a discrete-time Markov chain (DTMC)
  - With rates : a continuous-time Markov chain (CTMC)

- Probabilistic model checking can be used to verify the non-functional properties (quality attributes) of systems modelled in Event-B

Dependability-explicit probabilistic modelling

- Modelling faulty behaviour

- In the abstract model

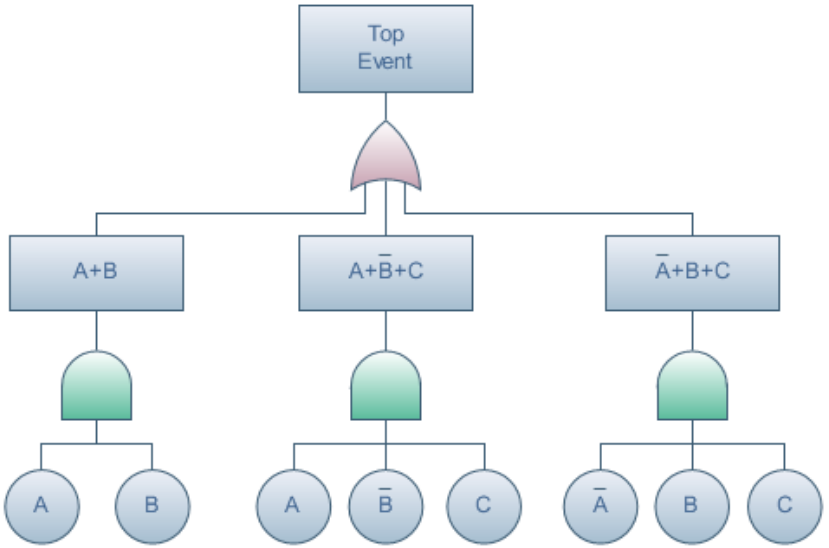
  result := result'. result': {OK_result, failure}

- In the probabilistic model

  result |⊕ OK_result @P ;failure @1-P

Assessing safety

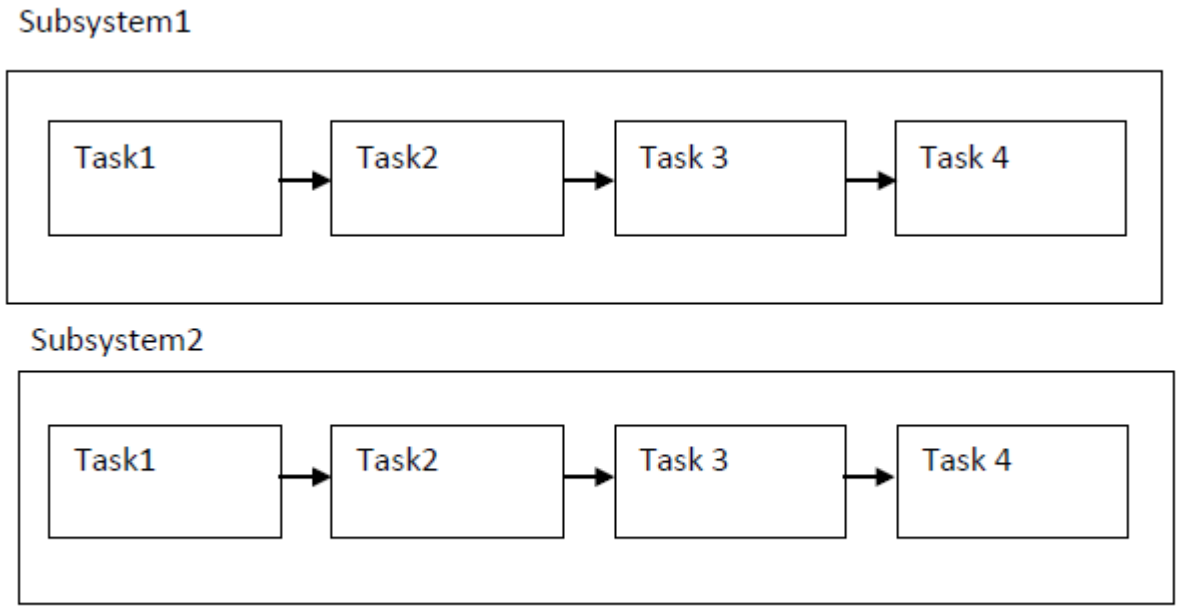
- Initially, a desired safety property is defined using abstract system variables
- We unfold it in the refinement until it refers to the basic system components
- At each refinement step we formulate gluing safety invariants that relate the newly introduced variables and abstract variables present in the safety property



At the final specification we have probabilistic model of the behaviour of each component and hence can calculate the probability of breaching safety

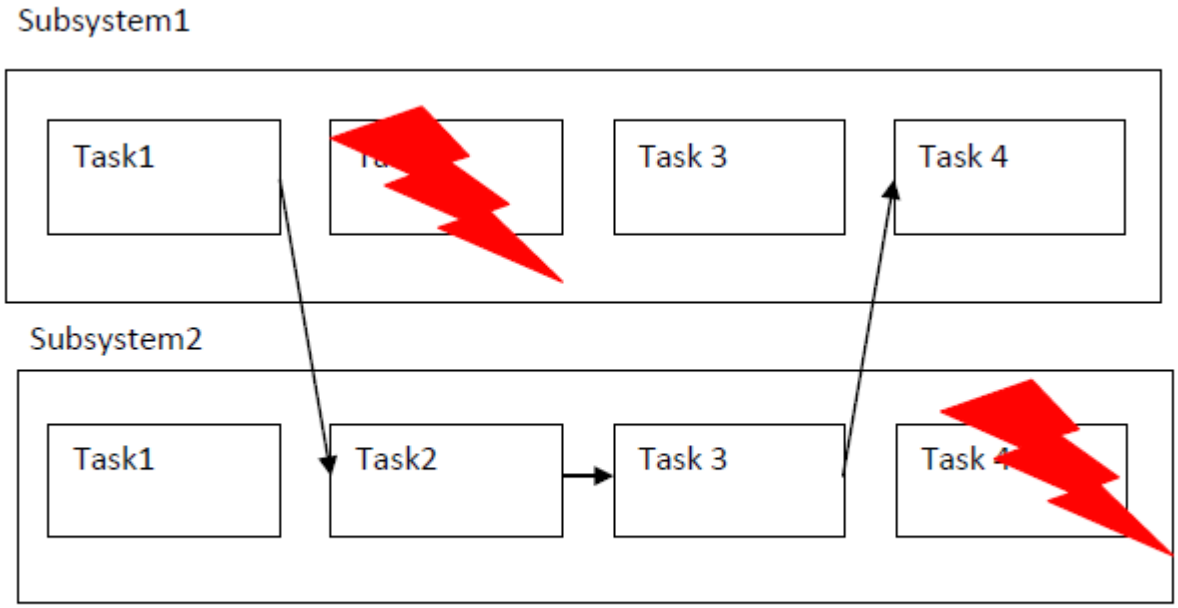
Essentially, we build a fault tree and demonstrate that the probability of a hazard occurrence is acceptably low

Modelling and assessing fault tolerant reconfigurable systems



- Subsystem 1 works until some component fails in it. Then the system switches to Subsystem2.
- Subsystem 2 works also until some component fails in it. But what when?
- Either need the 3rd subsystem or construct a system from “left-overs”

Modelling and assessing fault tolerant reconfigurable systems



- Reliability vs performance: are the target objectives reached?
- Integration with probabilistic analysis

  Allows us to assessed the derived reconfigurable system architecture and quantitatively verify that it achieves the desired reliability and performance objectives.

Service-oriented development: quantitative verification of QoS attributes

- What is the probability that at least one service execution will be aborted during a certain time interval?
- What is the probability that a number of aborted services during a certain time interval will not exceed some threshold?
- What is the mean number of served requests during a certain time interval?
- What is the mean number of aborted requests during a certain time interval?
- What is the mean number of failures of some particular subservice during a certain time interval?

Discussion

- Rich experience in modelling resilient systems from the transportation, aerospace and business information system domains

- Resilience-explicit modelling: two-fold approach
  - Creating modelling patterns and guidelines for representing and verifying certain dependability-related behaviour

  - Integrating (external) techniques for safety and reliability analysis into the formal development process of Event-B

FMEA worksheet fields

Component – name of a component

Failure mode – possible failure modes

Possible cause – possible cause of a failure

Local effects – caused changes in the component behaviour

System effect – caused changes in the system behaviour

Detection – determination of the failure

Remedial action – actions to tolerate the failure

Proposed FMEA worksheet fields

Global mode – name of a global mode

Failure mode – possible failure modes (unit failure)

Possible cause – possible cause of a failure

Local effects – caused changes in the component behaviour

System effect – caused changes in the system behaviour

Detection – determination of the failure

Remedial action – actions to tolerate the failure

The *general rule* of the rollback

- Mode Manager (MM) puts the system to the previous, however as advanced as possible, global mode where the failed unit is in *Off* state.

- All units that should be operational in the chosen degraded mode should be faul free
  - Otherwise, MM should put the system to a global mode where all failed units are in *Off* states.

FMEA worksheet for mode *Nominal*

Global mode	**Nominal**
Failure mode	GPS failure
Possible cause	Primary hardware failure
Local effects	Loss of precision of GPS
System effects	Switch to a degraded mode
Detection	Comparison of received data with the predicted one
Remedial action	If a failure occurs for the first time, then switch the nominal branch of the unit to the mode *Off* and the redundant branch of the unit to the mode *Coarse*. During the reconfiguration between the unit branches maintain the current global mode *Nominal*. If the redundant branch fails, then switch the branch to the mode *Off* and put the system to the previous, the most advanced, global mode where GPS unit is in *Off* state, i.e., to the mode *Safe*. Keep the unit status equals to *Locked* only if one of two branches is in *Coarse* state and there is no ongoing reconfiguration. Otherwise, change the unit status to *Unlocked.*

The results of integrating FMEA into the requirements engineering

- Allows for a systematic derivation of fault tolerance part of mode logic.
- Facilitates formalisation of the required conditions of mode consistency

Resilience in the context of service-oriented systems

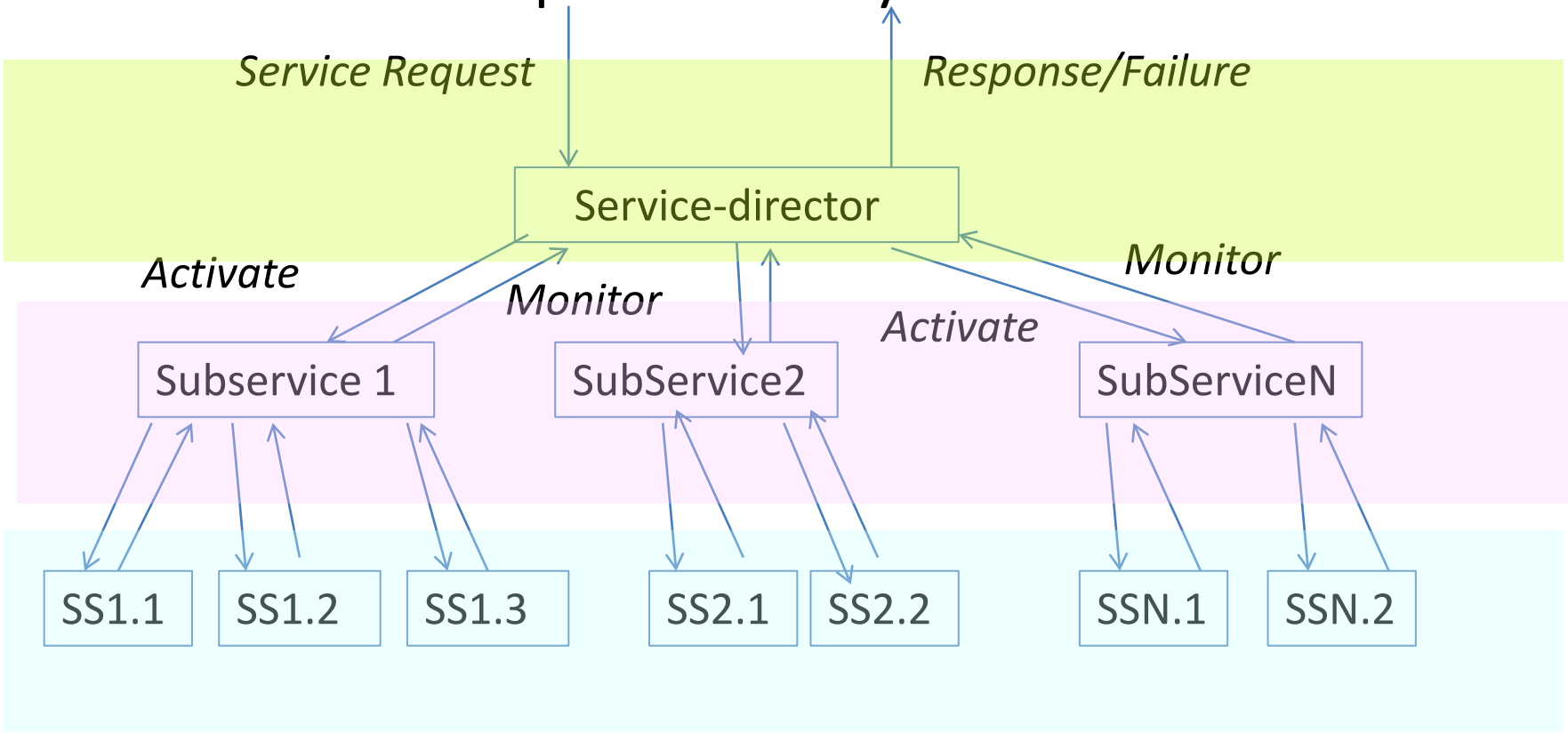
- Service-oriented computing enables rapid building of complex sofware by assembling readily-available services
- Formalisation of Lyra development approach (by Nokia) in Event-B: correctness and agility + fault tolerance and
- Are fault tolernace mechanisms appropriate, i.e., allow us to meet the desired quality of service (QoS) attributes?
- Need for techniques enabling evaluation of QoS attributes at early design stages

Proposed approach

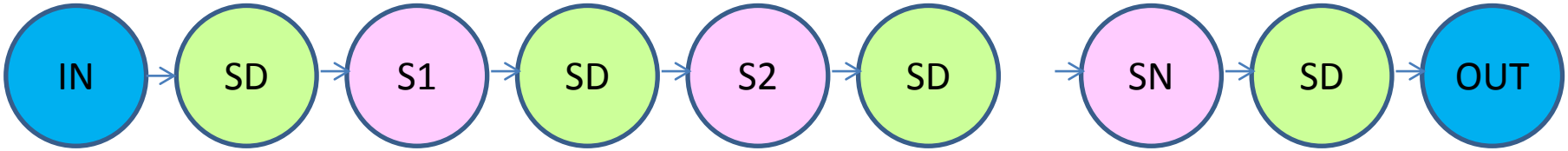
- Build model of dynamic service architecture in Event-B
- Formalise and verify dynamic service behaviour
- Augment Event-B model with stochastic information about rates and durations of the orchestrated services
- Use probabilistic model checking to verify the desired QoS attributes of the resultant Continuous-Time Markov Chain (CTMC)

Service-Oriented Systems (SOS)

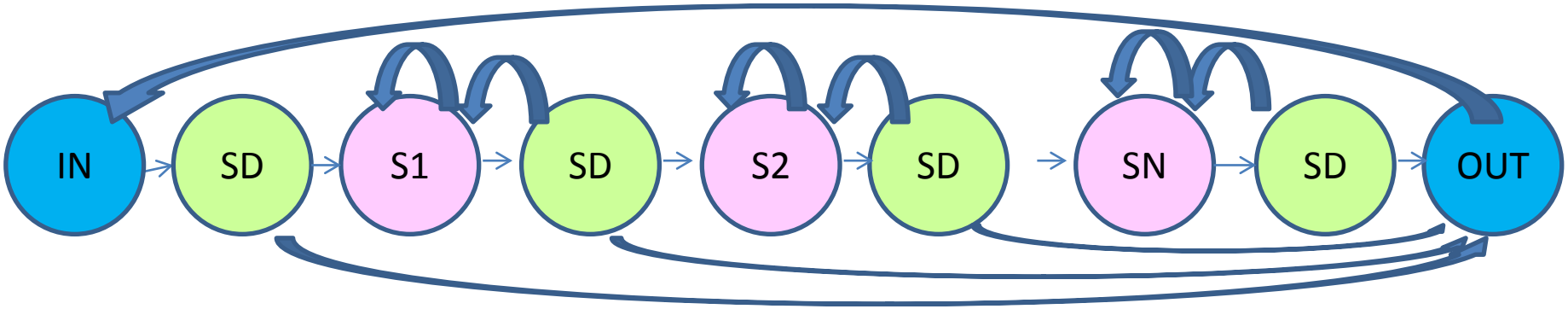
- Services are built by aggregating of lower-layer subservices
- Coordination is performed by *a service-director*



Flow of control



- Services are handled one by one
- After each subservice execution the service director might
  - allow the subservice to continue
  - proceed to the next subservice
  - retry subservice execution
  - abort (the entire service)



Event-B model as transition system

- Let Σ be a state space and, $\mathscr{E}$ a set of events, $\mathscr{I}$ invariant

- Event is defined as

  e = **when** G_e **then** R_e **end**

  can be seen as syntactic sugaring for

  $$e(\sigma, \sigma') = G_e(\sigma) \bigwedge R_e(\sigma, \sigma')$$

  - To define Event-B model as a transition system, we define functions *before(e)* and *after(e)*:

$$\text{before}(e) = \{\sigma \in \Sigma \mid \mathcal{I}(\sigma) \wedge G_e(\sigma)\}$$

$$\text{after}(e) = \{\sigma' \in \Sigma \mid \mathcal{I}(\sigma') \wedge (\exists \sigma \in \Sigma \cdot \mathcal{I}(\sigma) \wedge G_e(\sigma) \wedge R_e(\sigma, \sigma'))\}$$

Event-B model as transition system (cnt.)

- The behaviour of any Event-B machine is defined by a transition relation –>

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup_{e \in \mathcal{E}_\sigma} \text{after}(e)}{\sigma \rightarrow \sigma'}$$

where $\mathcal{E}_\sigma = \{e \in \mathcal{E} \mid \sigma \in \text{before}(e)\}$ is a subset of events enabled in σ

Formalising dynamic properties

- By defining Event-B specification we can formally define a number of essential dynamic properties of SoS under construction
- Formalisation of requirements can be added as a collection of model theorems

- If mapping between model events and “skeleton” is defined then the process can be automated

- Proving: either within Rodin platform or using external theorem provers

Probabilistic Event-B

- Transforming dynamic service architecture into a Continuous Time Markov Chain
- We aim at verifying time-bounded reachability and reward properties related to a possible abort of service execution
- Properties are specified as Continuous Stochastic Logic (CSL) formulae

Model transformation

- All events are augmented with the information about probability and duration of all the actions
- For the state $\sigma \in \Sigma$ and event $e \in \mathcal{E}$ where $\sigma \in before(e)$ assume that R_e can transform σ to a set of states $\{\sigma_1', \ldots, \sigma_m'\}$
- We augment every such transformation with a constant transition rate

$$\lambda_i \in \mathbb{R}^+,$$

- The sojourn time in state σ is exponentially distributed with parameter $\Sigma\lambda_i$
- Hereby we replace a nondeterministic choice between the possible successor states by the probabilistic choice associated with the exponential race conditions

Event-B model as a probabilistic transition system

- The behaviour of a probabilistically augmented Event-B machine is defined by a transition relation

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \xrightarrow{\Lambda} \sigma'}$$

where $\Lambda = \sum_{e \in \mathcal{E}_\sigma} \lambda_e(\sigma, \sigma')$

Quantitative verification of QoS attributes with PRISM

- What is the probability that at least one service execution will be aborted during a certain time interval?
- What is the probability that a number of aborted services during a certain time interval will not exceed some threshold?
- What is the mean number of served requests during a certain time interval?
- What is the mean number of aborted requests during a certain time interval?
- What is the mean number of failures of some particular subservice during a certain time interval?

Discussion

- Rich experience in modelling resilient systems from the transportation, aerospace and business information system domains

- Two types of approaches:
  - Focusing on creating modelling patterns and guidelines for representing and verifying certain resilience-related behavior
  - Integrating (external) techniques for safety and reliability analysis into the formal development process of Event-B

Challenges

- Scalability in formal modelling
- Powerful automatic tool support

- Event-B and Rodin platform:

  event-b.org
- Deploy project:

  http://www.deploy-project.eu/

Thank you!