



Elements of System Dynamics

Antoine B. Rauzy

PART 2. THE Σ^{TM} LANGUAGE

LECTURE 9. ADVANCED TOPICS

LECTURE 9. ADVANCED TOPICS

Key notions:

- Patterns and Metaphors
- Conflicting activities, priorities
- Functional Chains
- Pull and push modes
- Synchronous versus asynchronous models
- Differential equations
- Business Processes

Patterns and Metaphors

Using a powerful modeling language and efficient assessment algorithms is not sufficient to make the modeling process efficient. To make it efficient, one needs in addition a **methodology to design, document and maintain models**. With that respect, it is of primary importance to be able to reuse as much as possible modeling components within models and between models.

In languages such as Modelica or Matlab/Simulink, this goal is achieved via the design of **libraries of on-the-shelf ready-to-use modeling components**. Reusing components is also possible in Σ^{TM} , but to a much lesser extent. The reason is that system dynamics analyses considers systems at a high level of abstraction. Modeling components, except for very basic ones, tend thus to be specific to each system.

In Σ^{TM} , reuse is mostly achieved by the design of **modeling patterns**, i.e. **examples of models representing remarkable features** of the system under study. Once identified, patterns can be duplicated and adjusted for specific needs. To present and explain these patterns, one uses **metaphors**, i.e. illustrative examples stemmed from "daily life" that analysts and clients of systemic digital twins can easily understand.

Learning Objectives

This lecture first completes the presentation of Σ^{TM} .

Then it reviews some modeling patterns and metaphors that are often encountered in practice.

Agenda

Section 1. More on Activities

Section 2. Priorities

Section 3. Logistic and Production

Section 4. Differential Equations

Section 5. Business Processes

Section 6. Functional Chains

LECTURE 9. ADVANCED TOPICS

SECTION 1. MORE ON ACTIVITIES

Action at Start versus Duration

When an activity is launched, two instructions are executed in order:

- Its action at start;
- The calculation of its duration.

```
system ReactiveProducer
  int totalProduction(init = 0);
  int demand(init = 0);
  bool producing(init = true);
  activity ProduceOnDemand
    trigger:
      return not producing;
    start: {
      producing = true;
      demand = rangeDeviate(10, 20);
    }
    completion: {
      totalProduction += demand;
      producing = false;
    }
    duration:
      return 2*demand;
  end
end
```

ReactiveProducer



Lazy and Zealous Agents

```
system LazyAgent
  int remainingJobs(init = 3);
  activity TakeJob
    trigger: return remainingJobs>0;
    start: skip;
    completion: remainingJobs -= 1;
    duration: return 1;
  end
end
```

Lazy Agent Pattern:
3 successive jobs

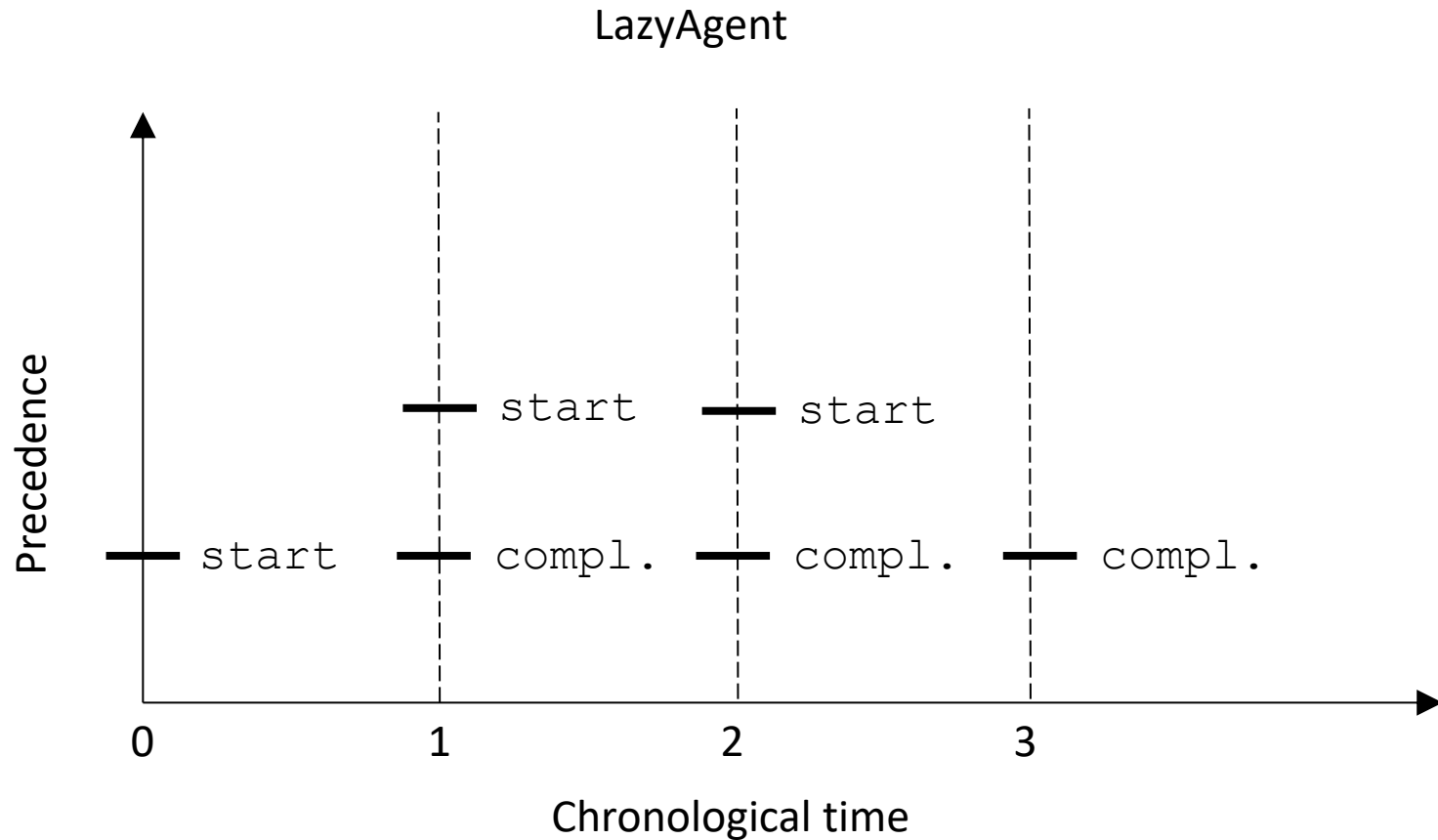
LazyAndZealousAgents

```
system ZealousAgent
  int remainingJobs(init = 3);
  activity TakeJob
    trigger: return remainingJobs>0;
    start: remainingJobs -= 1;
    completion: skip;
    duration: return 1;
  end
end
```

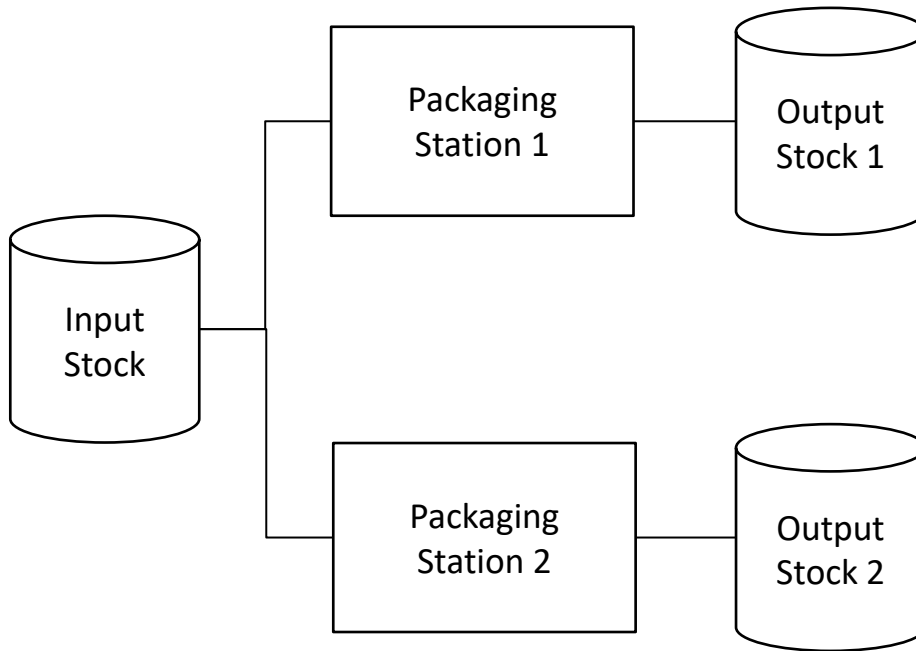


Zealous Agent Pattern:
3 jobs in parallel

Two-Dimensional Time



Conflicting Activities (1)



Two stations are packaging goods in parallel.

They take pre-processed goods in a shared stock and deliver packaged goods in their own stock.

The problem arises when it remains only one good in the input stock: which station will package it?

Conflicting Activities (2)

```
class PackagingStation
  bool packaging(init = false);
  int products(init = 0);
  virtual int inputProducts, outputProducts(init = 0);
  activity Package
    trigger:
      return not packaging and inputProducts>0;
    start: {
      inputProducts -= 1;
      products += 1;
      packaging = true;
    }
    completion: {
      packaging = false;
      products -= 1;
      outputProducts += 1;
    }
    duration:
      return 1;
  end
end
```

Treatment Station Pattern

Conflicting Activities (3)

```
system PackagingUnit
  int inputProducts(init = 3);
  PackagingStation Station1;
  PackagingStation Station2;
  int outputProducts1, outputProducts2(init = 0);
  actualizes Station1.inputProducts as inputProducts;
  actualizes Station1.outputProducts as outputProducts1;
  actualizes Station2.inputProducts as inputProducts;
  actualizes Station2.outputProducts as outputProducts2;
end
```

PackagingUnit

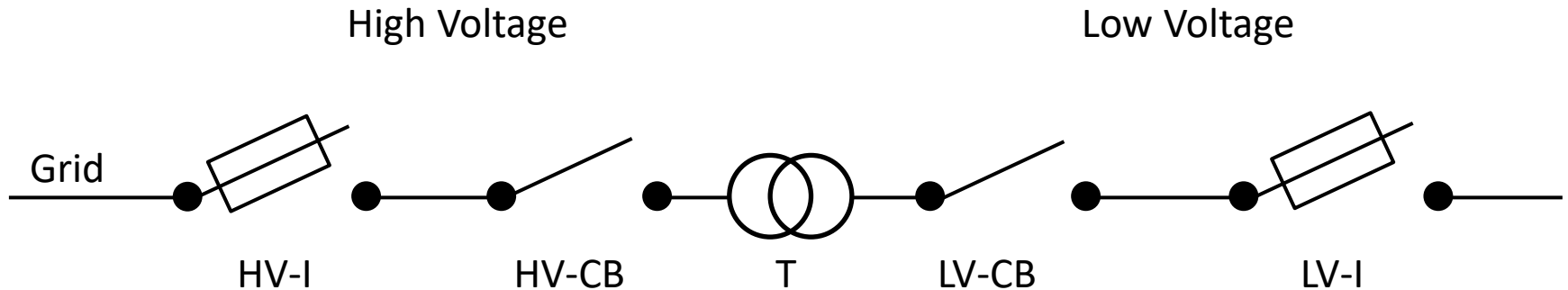


The activities `Package` of `Station1` and `Station2` are in conflict. The one that is started first is selected **at random**.

LECTURE 9. ADVANCED TOPICS

SECTION 2. PRIORITIES

Transformer Line



The transformer T transforms high voltage current (coming from the Grid) into low voltage current. To avoid a sustained electric arc that would damage the transformer, it is flanked by isolators HV-I and LV-I on the one hand, and the circuit breaker HV-CB and LV-CB on the other hand.

The order of operations isolators and circuit breakers are positioned matters.

Safe order:

To energize (turn on)

1. Close HV-I and LV-I
2. Close HV-CB
3. Close LV-CB

To de-energize (turn off)

1. Open LV-CB
2. Open HV-CB
3. Open HV-I and LV-I

Switches

```
domain SwitchState {OPEN, CLOSED}

class Switch
  SwitchState actualState(init = CLOSED);
  port SwitchState requestedState = CLOSED;
  port bool in = false;
  port bool out = in and actualState==CLOSED;
  bool switching (init = false);
end

activity Switch.Open
  trigger:
    return not switching and actualState==CLOSED and requestedState==OPEN;
  start:
    switching = true;
  completion: {
    actualState = OPEN;
    switching = false;
  }
  duration:
    return 0.01;
end
```

- Switch.Close designed along the same line
- Classes Isolator and CircuitBreaker derived from Switch
- Class Transformer just declares in and out ports

Controller

```
domain ControllerState {COMMAND, INHIBIT}

class Controller
  ControllerState state(init = COMMAND);
  port bool energize = phasing and state== COMMAND;
  bool phasing(init = false);
end

activity Controller.Command
  trigger:
    return not phasing and state== COMMAND;
  start:
    phasing = true;
  completion: {
    actualState = INHIBIT;
    phasing = false;
  }
  duration:
    return 100;
end
```

- Controller.Inhibit designed along the same line

Transformer Line (1)

```
system TransformerLine
```

```
  Isolator HVI;  
  CircuitBreaker HVCB;  
  Transformer T;  
  CircuitBreaker LVCB;  
  Isolator LVI;  
  Controller CTRL;
```

```
  port bool HVI.in = true;  
  port bool HVCB.in = HVI.out;  
  port bool T.in = HVCB.out;  
  port bool LVCB.in = T.out;  
  port bool LVI.in = LVCB.out;
```

```
  port SwitchState HVI.requestedState =  
    if CTRL.energize then CLOSED else OPEN;  
  port SwitchState HVCB.requestedState =  
    if CTRL.energize then CLOSED else OPEN;  
  port SwitchState LVCB.requestedState =  
    if CTRL.energize then CLOSED else OPEN;  
  port SwitchState LVI.requestedState =  
    if CTRL.energize then CLOSED else OPEN;
```

```
end
```

Problem: with this model, opening and closing of isolators and circuit-breakers can be done in any order.

Solution: give these activities priorities!

Priorities

```
activity Component.DoSomething
  trigger:
    return ...;
  start(priority = 10): {
    ...
  }
  completion(priority = 3): {
    ...
  }
  duration:
    return ...;
end
```

- Priorities are integer attributes of actions start and completion of activities.
- The action with a highest the priority is performed before the action with lowest priority.
- Default priority is 0.

Transformer Line (2)

```
system TransformerLine
...

// LVCB.Open -> HVCB.Open -> HVI.Open, LVI.Open
attribute HVI.Open.completion.priority = 2;
attribute HVCB.Open.completion.priority = 3;
attribute LVCB.Open.completion.priority = 4;
attribute LVI.Open.completion.priority = 2;

// HVI.Close, LVI.Close -> HVCB.Close -> LVCB.Close
attribute HVI.Close.completion.priority = 4;
attribute HVCB.Close.completion.priority = 3;
attribute LVCB.Close.completion.priority = 2;
attribute LVI.Close.completion.priority = 4;
end
```

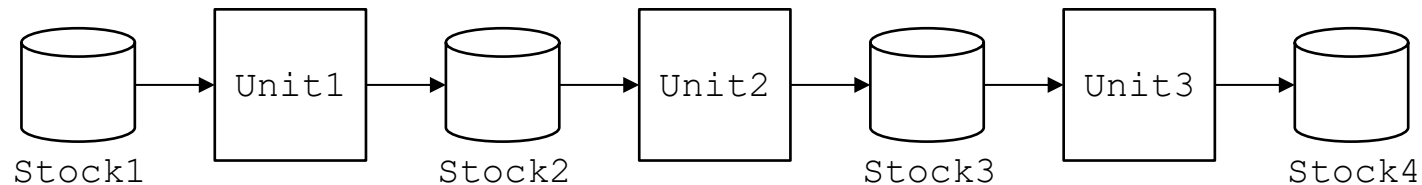
TransformerLine



LECTURE 9. ADVANCED TOPICS

SECTION 3. LOGISTIC AND PRODUCTION PATTERNS

Logistic and Production Systems



Goods are treated/transported along the chain.

Four fundamental modeling patterns:

- Goods are followed individually along the chain.
- Levels of stocks are followed; each treatment and transport activity has its own duration.
- Levels of stocks are followed; all treatment and transport activities are synchronized, e.g. monthly.
- Levels of production are followed; Only changes in treatment and transport levels are traced, not the treatment and transport activities themselves

Concrete

Abstract

The "right" level of abstraction depends on what we want to learn about the system.

LECTURE 9. ADVANCED TOPICS

SECTION 3.1. CUSTOM-BUILT ASSEMBLY LINE PATTERN

Geostationary Satellite Production

The production of geostationary satellites can be divided into 5 steps, each taking place in a separated station:



Station/Phase	Duration Range	Percentage (Approx.)	Variation Factor
I. Bus & Subsystem Assembly	2 - 6 Months	10% - 20%	Complexity of the structure, number of components, and initial integration issues.
II. Payload Integration	1 - 4 Months	5% - 15%	Complexity of the payload, need for precise alignment, and number of instruments.
III. Functional Testing (End-to-End)	1 - 3 Months	10% - 15%	Length of test scripts, time required for troubleshooting software/electrical anomalies.
IV. Environmental Testing (TVAC, Vibe)	4 - 12 Months	50% - 70%	This is the main driver. The satellite must cycle multiple times in the Thermal Vacuum (TVAC) chamber. A single TVAC run can take 4-8 weeks.
V. Final Preparation & Launch Campaign	1 - 2 Months	5% - 10%	Logistics, transfer to the launch site, final checks, and encapsulation.

Because it is a custom-built production, it may be worth to follow the production of each satellite individually.

Satellite

```
class Satellite
  int phase(init = 1);
  bool active(init = false);
  virtual int previousSatellitePhase(init = 6);
end

activity Satellite.BusAndSubsystemAssembly
  trigger:
    return phase==1 and not active and phase<previousSatellitePhase;
  start:
    active = true;
  completion: {
    phase += 1;
    active = false;
  }
  duration:
    return uniformDeviate(2, 6);
end
```

Ensures that the current phase cannot be started before the previous satellite in production has not completed this phase.

The other production phases are represented in the same way.

Satellite Factory

```
system SatelliteFactory
  Satellite S1;
  Satellite S2;
  Satellite S3;
  Satellite S4;
  Satellite S5;
  Satellite S6;
  Satellite S7;
  Satellite S8;
  Satellite S9;
  Satellite S10;
  int finalPhase(init = 6);
  actualizes S1.previousSatellitePhase as finalPhase;
  actualizes S2.previousSatellitePhase as S1.phase;
  actualizes S3.previousSatellitePhase as S2.phase;
  actualizes S4.previousSatellitePhase as S3.phase;
  actualizes S5.previousSatellitePhase as S4.phase;
  actualizes S6.previousSatellitePhase as S5.phase;
  actualizes S7.previousSatellitePhase as S6.phase;
  actualizes S8.previousSatellitePhase as S7.phase;
  actualizes S9.previousSatellitePhase as S8.phase;
  actualizes S10.previousSatellitePhase as S9.phase;
end
```

Although this modeling pattern "works", it is not recommended to use Σ^{TM} for this type of systems. Agent-based modeling frameworks may be more appropriate.

If you nevertheless use this pattern, think to generate models with a script (see Lecture 11).

SatelliteFactory



LECTURE 9. ADVANCED TOPICS

SECTION 3.2. PRODUCTION PATTERNS

Push and Pull Mode



	Upstream → Downstream	Upstream ← Downstream
Supplier	SupplyRawMaterial	
Producer	ProduceProducts	OrderRawMaterial
Consumer	ConsumeProducts	OrderProducts

- In **push mode**, only the upstream to downstream flow is represented.
- In **pull mode**, both the upstream to downstream flow and the downstream to upstream flow are represented (which requires to add stocks of orders).

Synchronous versus Asynchronous



	Upstream → Downstream	Upstream ← Downstream
Supplier	SupplyRawMaterial	
Producer	ProduceProducts	OrderRawMaterial
Consumer	ConsumeProducts	OrderProducts

- In a **synchronous** model, all activities are performed simultaneously and have the same duration. Flow may vary stochastically.
- In **asynchronous** model, each activity has its own duration. Both flows and durations may vary stochastically.

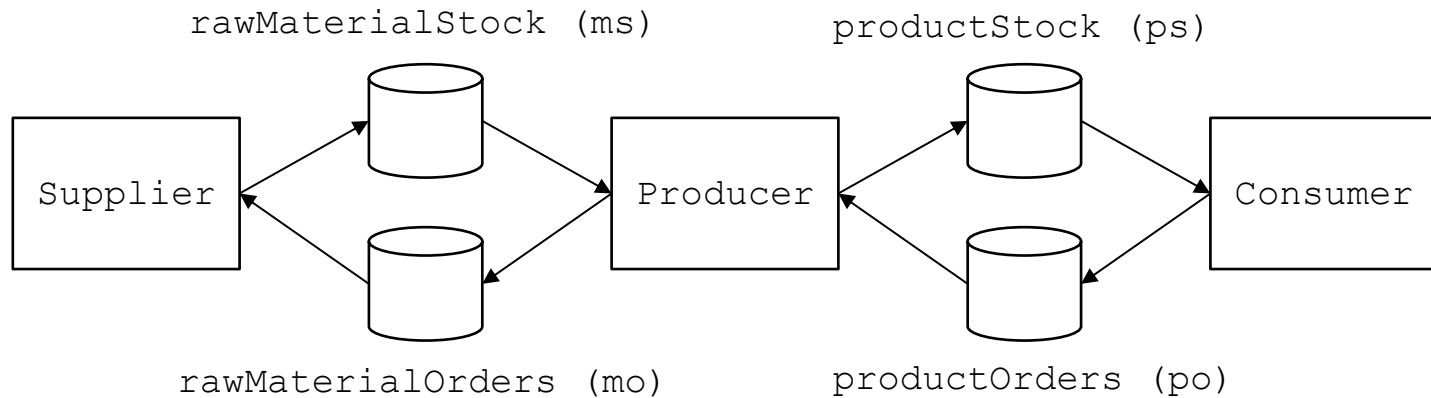
Kanban

Kanban (Japanese: *かんばん* meaning signboard) is a scheduling system for lean manufacturing (also called just-in-time manufacturing, abbreviated JIT). Taiichi Ohno at Toyota, developed kanban to improve manufacturing efficiency. The system takes its name from the cards that track production within a factory.

Toyota has formulated six rules for the application of kanban:

- Each process issues requests (kanban) to its suppliers when it consumes its supplies.
- Each process produces according to the quantity and sequence of incoming requests.
- No items are made or transported without a request.
- The request associated with an item is always attached to it.
- Processes must not send out defective items, to ensure that the finished products will be defect-free.
- Limiting the number of pending requests makes the process more sensitive and reveals inefficiencies.

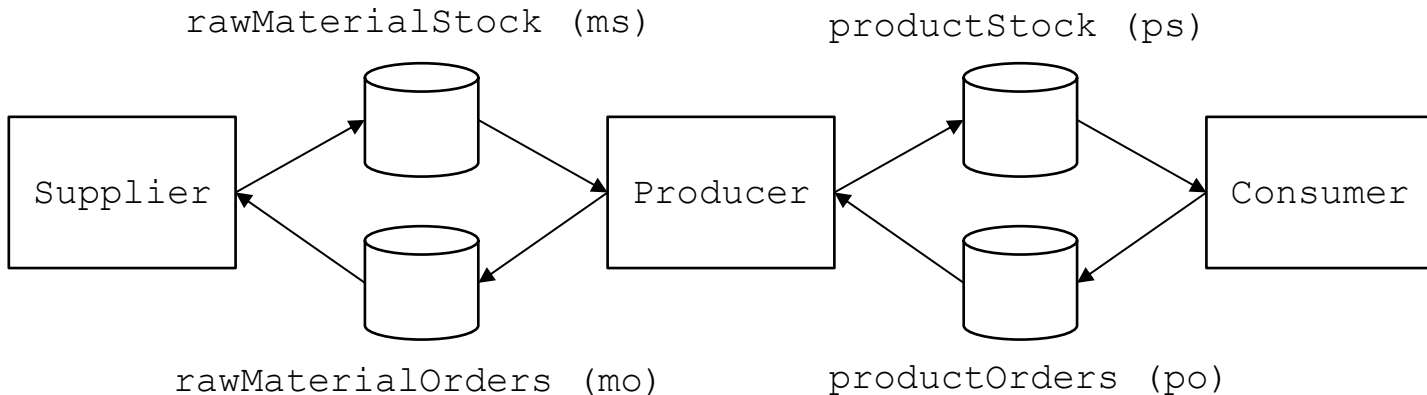
Asynchronous Kanban Pattern



Assumptions:

- The producer needs 2 tons of raw material to produce 1 ton of products.
- The consumer consumes at rate of 1 ton product per day, consequently the producer produces products at a rate of 1 ton per day and consumes raw material at a rate of 2 tons per day, while the supplier supplies raw material at a rate of 2 tons of per day.
- The consumer consumes by batches of 2 tons (in 2 days).
- The consumer orders products once a week (5 days). It takes him 1 day to send an order.
- The producer produces products by batches of 10 tons (in 2 weeks).
- The producer orders raw material once in 5 weeks. It takes him 1 day to send an order.
- The supplier supplies raw material by batches of 30 tons (in 3 weeks).

Asynchronous Kanban Pattern



Activity	Trigger	Start	Completion	Duration
Supplier Supply	$mo \geq ms$		$mo -= \min(30, mo)$ $ms += 30$	15
Producer Produce	$po \geq ps$ $ms \geq 20$	$ms -= 20$	$po -= \min(10, po)$ $ps += 10$	10
Producer Order	$po \geq ps$ $ms < 20$ $mo < 50$		$mo += 50$	1
Consumer Consume	$ps \geq 2$	$ps -= 2$		2
Consumer Order	$ps < 2$ $po < 5$		$po += 5$	1

Producer

```
class Producer
  virtual int rawMaterialStock(init = 0);
  virtual int rawMaterialOrders(init = 0);
  virtual int productStock(init = 0);
  virtual int productOrders(init = 0);
  bool producing(init = false);
  bool ordering(init = false);
end

activity Producer.Produce
  trigger:
    return not producing and productOrders>productStock and
      rawMaterialStock>=20;
  start: {
    rawMaterialStock -= 20;
    producing = true;
  }
  completion: {
    productOrders -= min(10, productOrders);
    productStock += 10;
    producing = false;
  }
  duration:
    return 20;
end
```

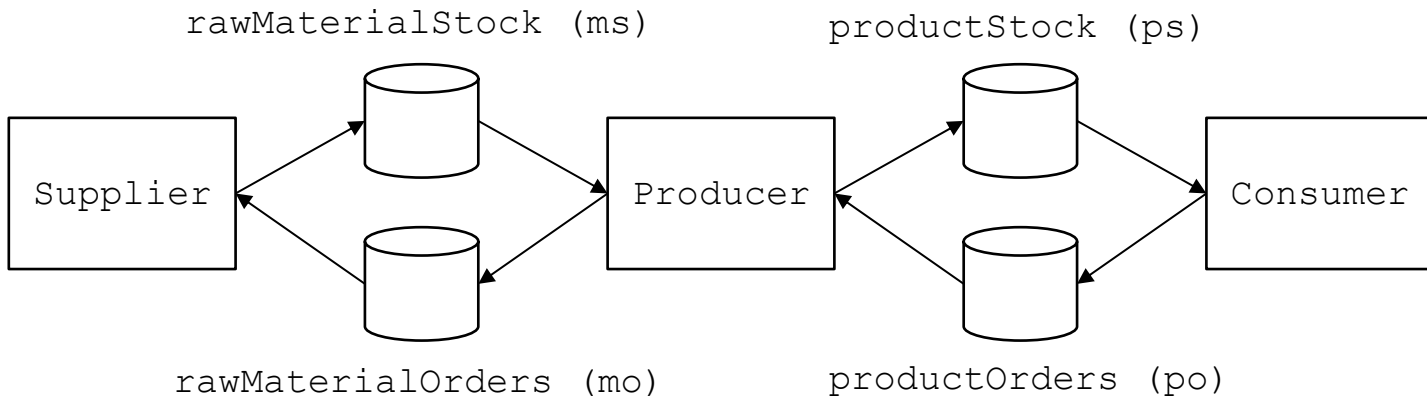
Asynchronous Kanban

```
system AsynchronousKanban
  Supplier S;
  Producer P;
  Consumer C;
  int rawMaterialStock(init = 0);
  int rawMaterialOrders(init = 0);
  int productStock(init = 0);
  int productOrders(init = 0);
  actualizes S.rawMaterialStock as rawMaterialStock;
  actualizes S.rawMaterialOrders as rawMaterialOrders;
  actualizes P.rawMaterialStock as rawMaterialStock;
  actualizes P.rawMaterialOrders as rawMaterialOrders;
  actualizes P.productStock as productStock;
  actualizes P.productOrders as productOrders;
  actualizes C.productStock as productStock;
  actualizes C.productOrders as productOrders;
end
```

AsynchronousKanban



Synchronous Kanban Pattern



Activity	Trigger	Start	Completion	Duration
Supplier Supply	$mo \geq ms$		$mo -= \min(60, mo)$ $ms += 60$	30
Producer Produce	$po \geq ps$ $ms \geq 60$	$ms -= 60$	$po -= \min(30, po)$ $ps += 30$	30
Producer Order	$po \geq ps$ $ms < 60$ $mo < 60$		$mo += 60$	1
Consumer Consume	$ps \geq 30$	$ps -= 30$		30
Consumer Order	$po < 30$		$po += 30$	1

Discussion

Synchronous models are easier to design than asynchronous ones. They stand at a higher level of abstraction, as the durations of their key activities are the "least common multiple" of actual durations. Synchronous models are better suited for mass production (or logistic) systems.

Synchronous and asynchronous models manage uncertainties differently, e.g.

	Quantity	Duration
Synchronous	<code>normalDeviate(30, 2)</code>	30
Asynchronous	10	<code>triangularDeviate(9, 12, 10)</code>

LECTURE 9. ADVANCED TOPICS

SECTION 3.3. CAPACITY-ORDER-PRODUCTION PATTERN

Crude Oil Refining

Crude oil refining is a continuous flow process where intermediate products must be processed immediately to prevent storage overflow or process cooling, which can be detrimental to the product quality.

Stage	Treatment Unit	Primary Function	Capacity Fluctuation Cause
1	Crude Distillation Unit (CDU)	Separates crude oil into fractions based on boiling points.	Changes in the inlet crude oil composition, fouling, or pump/heater maintenance.
2	Hydrotreater Hydrodesulfurizer (HDS)	Removes sulfur and other contaminants from a specific fraction.	Catalyst activity degradation over time, hydrogen compressor performance, or scheduled catalyst regeneration.
3	Catalytic Reformer (RFM)	Restructures the molecular components of a fraction (e.g., naphtha) to produce high-octane gasoline components (reformate).	Catalyst coke build-up (requires frequent regeneration), heater/furnace temperature constraints, or changes in target octane number (which affects reaction severity).

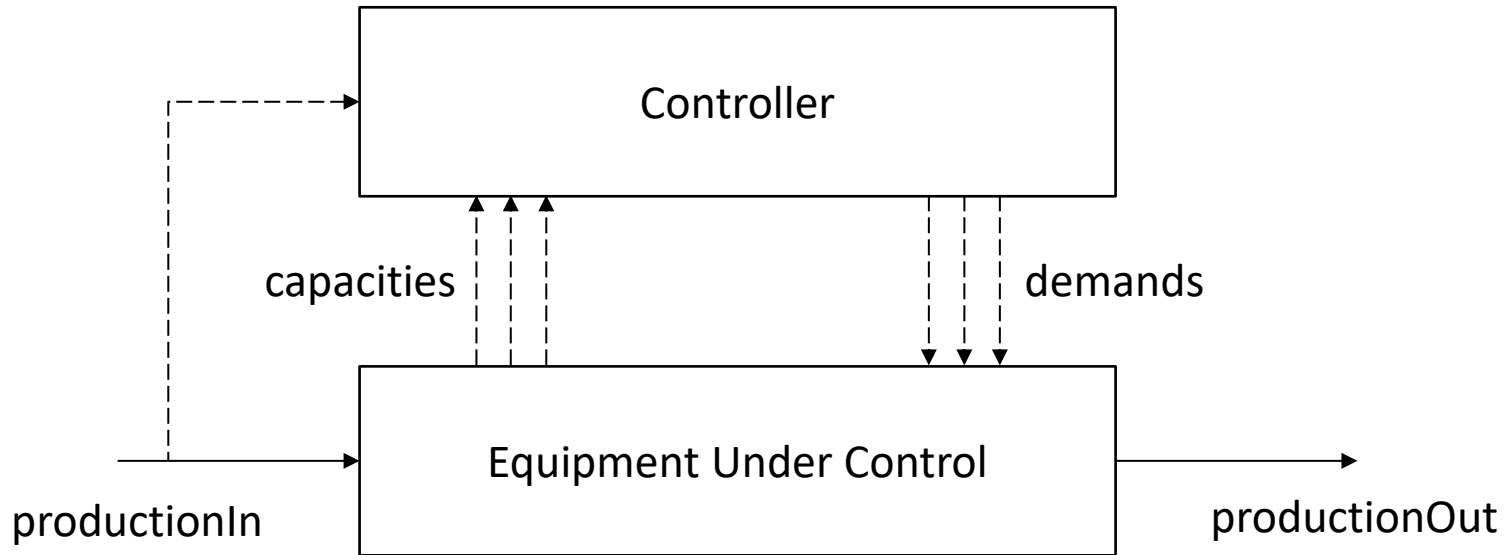
Assumptions*

- The capacity of treatment units degrades through time from a maximum to a minimum. When the minimum is reached, a maintenance operation is performed that takes back the capacity to its maximum.
- At any time, the production is the minimum of the capacities of treatment units.
- The degradation of treatment units proceeds by discrete jumps. The time between two successive jumps is exponentially distributed. The distribution of the size of jumps depends on the components.
- During maintenance operations, treatment units are shut down. The maintenance times are uniformly distributed between a lower and an upper bounds.

Data	CDU	HDS	RFM
Maximum Capacity	100	105	95
Minimum Capacity	85	85	85
Degradation Rate	$6.6 \cdot 10^{-2} \text{ d}^{-1}$	$1.6 \cdot 10^{-2} \text{ d}^{-1}$	$3.3 \cdot 10^{-2} \text{ d}^{-1}$
Size of Jumps	Uniform(1, 3) d	Uniform(3, 7) d	Uniform(2, 6) d
Maintenance Time	Uniform(3, 7) d	Uniform(7, 21) d	Uniform(10, 14) d

(*) These assumptions are oversimplifying the description of the refining process.

Capacity/Demand/Production Pattern



- The units of the Equipment Under Control transmit their capacities to the Controller.
- Considering these capacities and upstream production, the Controller emits demands to the units.
- Each unit adjusts its production to the production it receives, its capacity and its demand.

Treatment Units (1)

```
domain TreatmentUnitState {STANDBY, OPERATION, SHUTDOWN, MAINTENANCE}

class TreatmentUnit
  parameter float capacityMinimum = 80;
  parameter float capacityMaximum = 100;
  parameter float degradationRate(unit = "d-1") = 1.0e-2;
  parameter float degradationSizeMinimum = 1;
  parameter float degradationSizeMaximum = 3;
  parameter float maintenanceTimeMinimum(unit = "d") = 5;
  parameter float maintenanceTimeMaximum(unit = "d") = 10;
  TreatmentUnitState state(init = STANDBY);
  float capacity(init = capacityMaximum);
  port float demand = 0;
  port float productionIn = 0;
  port float productionOut = min(capacity, demand, productionIn);
  port bool overflow = productionOut < productionIn;
end
```

Treatment Units (2)

```
activity TreatmentUnit.Operation
  float operationTime;
  trigger:
    return state==STANDBY;
  start: {
    operationTime = exponentialDeviate(degradationRate);
    state = OPERATION;
  }
  completion: {
    capacity -= uniformDeviate(degradationSizeMinimum,
                               degradationSizeMaximum);
    if capacity < capacityMinimum then {
      state = SHUTDOWN;
      capacity = 0;
    }
    else
      state = STANDBY;
  }
  duration:
    return operationTime;
end
```

The activity Maintenance is defined in a similar way.

Crude Distillation Unit

```
class CrudeDistillationUnit
  extends TreatmentUnit;
  parameter float capacityMinimum = 85;
  parameter float capacityMaximum = 100;
  parameter float degradationRate(unit = "d-1") = 6.6e-2;
  parameter float degradationSizeMinimum = 1;
  parameter float degradationSizeMaximum = 3;
  parameter float maintenanceTimeMinimum(unit = "d") = 3;
  parameter float maintenanceTimeMaximum(unit = "d") = 7;
end
```

The classes for Hydrodesulfurizers and Catalytic Reformers for are defined in a similar way.

Crude Oil Refinery

```
system CrudeOilRefinery
  CrudeDistillationUnit CDU;
  Hydrodesulfurizer HDS;
  CatalyticReformer RFM;
  system Controller
    port float capacity = min(main.CDU.capacity, main.HDS.capacity,
      main.RFM.capacity);
    port float demand = capacity;
  end
  port float CDU.demand = Controller.demand;
  port float HDS.demand = Controller.demand;
  port float RFM.demand = Controller.demand;
  port float CDU.productionIn = Controller.demand;
  port float HDS.productionIn = CDU.productionOut;
  port float RFM.productionIn = HDS.productionOut;
  port float production = RFM.productionOut;
  observer productionMean = timedMean(production);
  observer productionTotal = timedSum(production);
end
```

CrudeOilRefinery



We assume here that the the upstream production is adjusted to the capacity.

LECTURE 9. ADVANCED TOPICS

SECTION 4. DIFFERENTIAL EQUATIONS

Back on Synchronous Models

In synchronous models, all activities are performed simultaneously. Up to some ordering issues, there could be only one activity, Step. The reason why one maintains several activities is to have a better understanding of the behavior of the system, just as one decomposes the system into a hierarchy of subsystems for the very same purpose.

If the set of variables involved in the model is $V = \{v_1, \dots, v_n\}$, the next step calculation is as follows.

$$\begin{array}{ccc} v_1 & = & f_1(V) \\ \vdots & & \vdots \\ v_n & = & f_n(V) \end{array} \quad \text{where the } f_i\text{'s may be stochastics.}$$

Problem: if v_i is updated before v_j and f_j depends on v_i , the calculation may go wrong.

Population Dynamics Metaphor

The **Leslie model** is a discrete, age-structured model of population dynamics. Consider a rabbit population in a given ecosystem. We can divide this population into three groups:

- Juveniles (0-1 years)
- Young adults (1-2 years)
- Adults (2-3 years).

Rabbits older than 3 years are assumed negligible.

Biological parameters:

- Fertility (average newborns per year)
 - Juveniles: $f_1 = 0$
 - Young adults: $f_2 = 3.5$
 - Adults: $f_3 = 1.5$
- Survival rates (per year)
 - Juveniles $\rightarrow$ young adults: $s_1 = 0.25$
 - Young adults $\rightarrow$ adults: $s_2 = 0.50$

Population dynamics:

$$P_{t+1} = \begin{pmatrix} f_1 & f_2 & f_3 \\ s_1 & 0 & 0 \\ 0 & s_2 & 0 \end{pmatrix} \cdot P_t$$

Leslie matrix

Rabbit Population

```
system RabbitPopulation
  float juveniles(init = 100);
  float youngAdults(init = 100);
  float adults(init = 100);
  bool stepping(init = false);
end
```

```
activity RabbitPopulation.Step
  trigger: return not stepping;
  start:
    stepping = true;
  completion: {
    float juvenilesPrevious, youngAdultsPrevious, adultsPrevious;
    juvenilesPrevious = juveniles;
    youngAdultsPrevious = youngAdults;
    adultsPrevious = adults;
    juveniles = 3.5*youngAdultsPrevious + 1.5*adults;
    youngAdults = 0.25*juveniles;
    adults = 0.50*youngAdults;
    stepping = false;
  }
  duration: return 1.0;
end
```

RabbitPopulation



Variables are created to store previous values of the state variables

Finite Difference Equations

A **finite difference** is a way to approximate the rate of change (derivative) of a function using values at discrete, equally spaced points. Given a function φ , we can define the difference operator $\Delta[f] = \varphi(n + 1) - \varphi(n)$.

Let $x(t)$ be a unknown function from non-negative reals to reals. Assume that the following quantities are known.

- $x_0 = x(0)$
- $\Delta[f] = f(x(n), n)$

Then we can calculate the value of x at any point in time using the recurrence:

- $x_{n+1} = x_n + \Delta[f]$

This extends to systems of difference equations, such as the Leslie model. It is tempting to extend this principle to solve numerically systems of **differential equations**. However, this is in general not possible.

The reason is the **extreme sensitivity** of the latter to **numerical approximations and errors**. The Leslie model shows already such sensitivity.

Compartmental models

Compartmental models are a mathematical framework used to simulate how populations move between different states or "compartments". In mathematical modelling of infectious diseases, the population is typically divided into compartments labeled with shorthand notation – most commonly S, I, and R, representing Susceptible, Infectious, and Recovered individuals [Wikipedia]. The evolution of the population in these three compartment is governed by the following different equations.

$$\begin{aligned}\frac{dS}{dt} &= -\frac{\beta}{N}IS \\ \frac{dI}{dt} &= \frac{\beta}{N}IS - \gamma I \\ \frac{dR}{dt} &= \gamma I\end{aligned}\quad \text{where } \left\{ \begin{array}{l} N = S + I + R \\ \beta \text{ is the infection rate (per day)} \\ \gamma \text{ is the recovery rate (per day)} \end{array} \right.$$

Approximating differential equations by discretizing them and using the same pattern as before, we can study (numerically) the evolution of disease.

Forward Euler Method

The **forward Euler method** is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value.

Let $x(t)$ be a unknown function from non-negative reals to reals. Assume that the following quantities are known.

- $x_0 = x(0)$
- $\frac{dx}{dt} = f(x(t), t)$

Then for a sufficiently small h , we can approximate the value of x at $t = N \times h$ by calculating the series:

$$x_{n+1} = x_n + h \times f(x_n, t)$$

This extends to systems of differential equations.

SIR Model: Forward Euler Method

```
activity SIR.FEM.Step
  trigger:
    return not stepping;
  start:
    stepping = true;
  completion: {
    float t, x, y;
    t = 0;
    while t < T {
      x = (beta/N) * S * I * dt;
      y = gamma * I * dt;
      S += -x;
      I += x - y;
      R += y;
      t += dt;
    }
    stepping = false;
  }
  duration:
    return T;
end
```



Runge-Kutta Method(s)

Although extremely simple and sometimes efficient, the Forward Euler Method shows some numerical instability. **Runge-Kutta** methods aim at being more stable. The most widely used method is the explicit Runge-Kutta method of order 4. The step calculation goes as follows.

$$y_{n+1} = y_n + \frac{h}{6} \times (k_1 + 2k_2 + 2k_3 + k_4)$$
$$\begin{aligned} k_1 &= f(y_n, t_n) \\ k_2 &= f(y_n + h \frac{k_1}{2}, t_n + \frac{h}{2}) \\ k_3 &= f(y_n + h \frac{k_2}{2}, t_n + \frac{h}{2}) \\ k_4 &= f(y_n + hk_3, t_n + h) \end{aligned}$$

It is possible to apply this method to the SIR example. However, one must add a normalization step to ensure that the sum $S + I + R$ stays constant through the iterations.

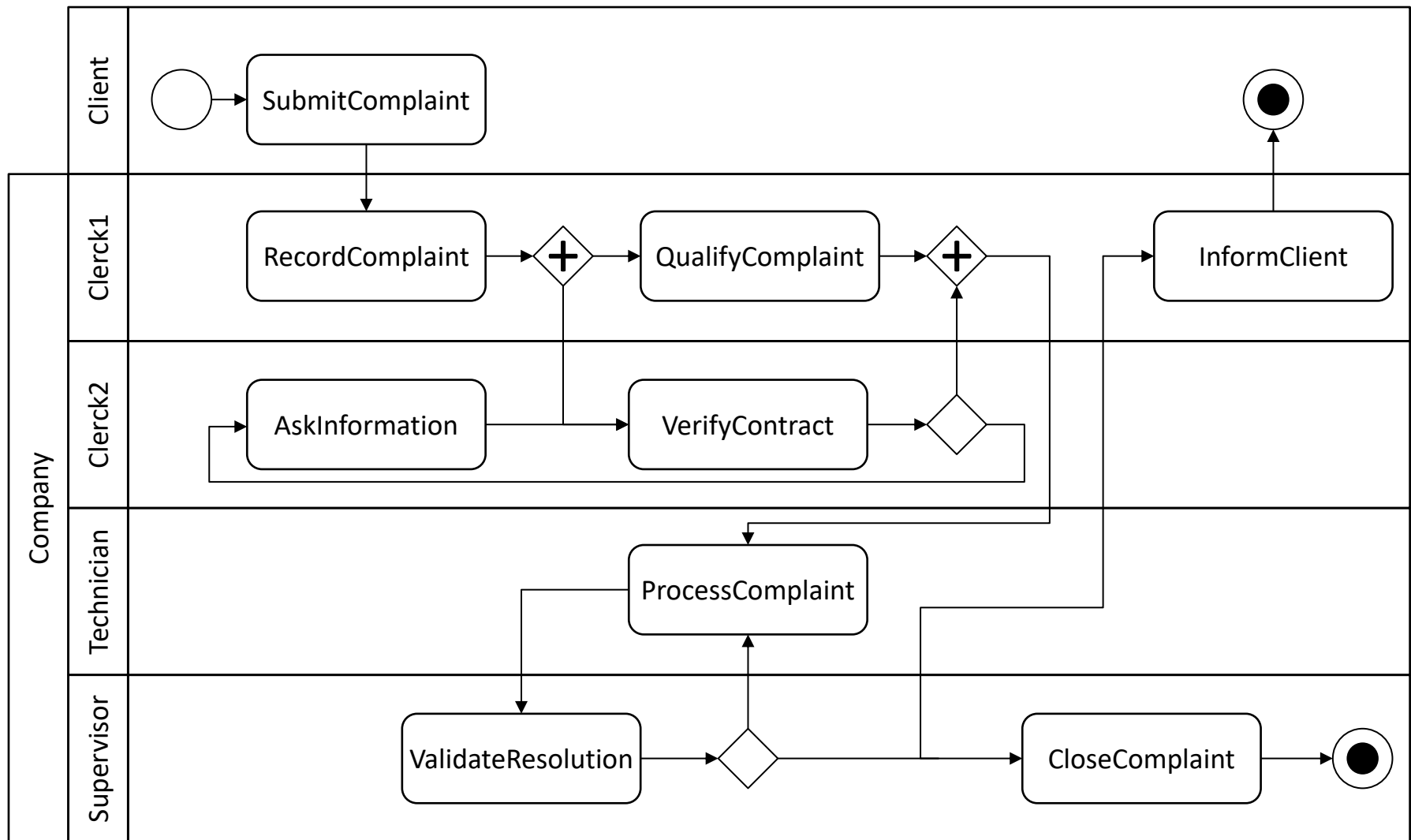


In a word, Σ^{TM} is not designed to solve systems of differential equations.

LECTURE 9. ADVANCED TOPICS

SECTION 5. BUSINESS PROCESSES

Customer Complaint Handling Process



Customer Complaint Handling Data

The duration of all task can be described by means of a triangular distribution (in minutes)

Task	Low	High	Expected
SubmitComplaint	10	30	15
RecordComplaint	2	10	5
QualifyComplaint	5	20	10
VerifyContract	3	15	8
AskInformation	5	20	10
ProcessComplaint	10	60	30
ValidateResolution	2	10	5
InformClient	2	10	5
CloseComplaint	3	6	4

The customer contract is completed in 80% of the cases.

The supervisor validates the resolution in 90% of the cases.

Tasks

```
domain TaskState {TODO, DOING, REDO, DONE}
```

When the task is not completed or failed

```
class Task
```

```
  parameter float minimumDuration(unit = "m") = 0;
```

```
  parameter float maximumDuration(unit = "m") = 2;
```

```
  parameter float expectedDuration(unit = "m") = 1;
```

```
  TaskState state(init = TODO);
```

```
  port bool ready = false;
```

```
end
```

True if and only if all preceding tasks are completed

```
activity Task.Do
```

```
  trigger:
```

```
    return ready and (state==TODO or state==REDO);
```

```
  start:
```

```
    state = DOING;
```

```
  completion:
```

```
    state = DONE;
```

```
  duration:
```

```
    return triangularDeviate(minimumDuration, maximumDuration,  
                             expectedDuration);
```

```
end
```

Project

```
system CustomerComplaint
  system Client
    Task SubmitComplaint
      parameter float minimumDuration = 10;
      parameter float maximumDuration = 30;
      parameter float expectedDuration = 15;
    end
  end
  system Company
    system Clerk1 ... end
    system Clerk2 ... end
    system Technician ... end
    system Supervisor ... end
  end

  port bool Client.SubmitComplaint.ready = true;
  port bool Company.Clerk1.RecordComplaint.ready = Client.SubmitComplaint.state==DONE;
  port bool Company.Clerk1.QualifyComplaint.ready = Company.Clerk1.RecordComplaint.state==DONE;
  ...
end
```

Instantiation of tasks

Precedence constraints

CustomerComplaint



Fallible Tasks

```
...
system Clerk2
  Task VerifyContract
    parameter float minimumDuration = 3;
    parameter float maximumDuration = 15;
    parameter float expectedDuration = 8;
    parameter float failureProbability = 0.20;
  end

...
port bool Company.Clerk2.VerifyContract.ready =
  (Company.Clerk2.VerifyContract.state==TODO and Company.Clerk1.RecordComplaint.state==DONE)
or (Company.Clerk2.VerifyContract.state==REDO and Company.Clerk2.AskInformation.state==DONE);

...
activity Company.Clerk2.VerifyContract.Do
  completion: {
    if uniformDeviates(0.0, 1.0) < failureProbability then {
      main.Company.Clerk2.AskInformation.state = TODO;
      state = REDO;
    }
    else state = DONE;
  }
end

...
```

First attempt

Following attempts

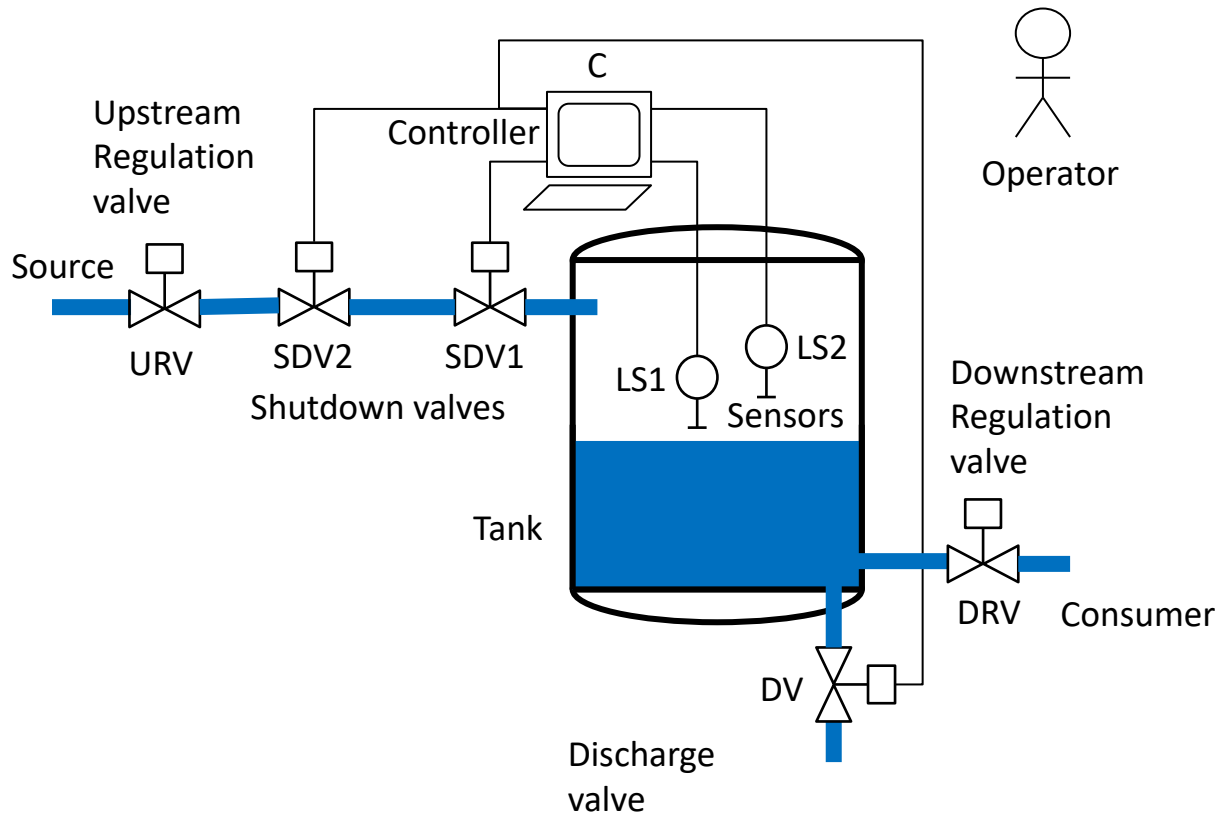
When the task fails, all tasks in its loop are reset

LECTURE 9. ADVANCED TOPICS

SECTION 6. FUNCTIONAL CHAINS

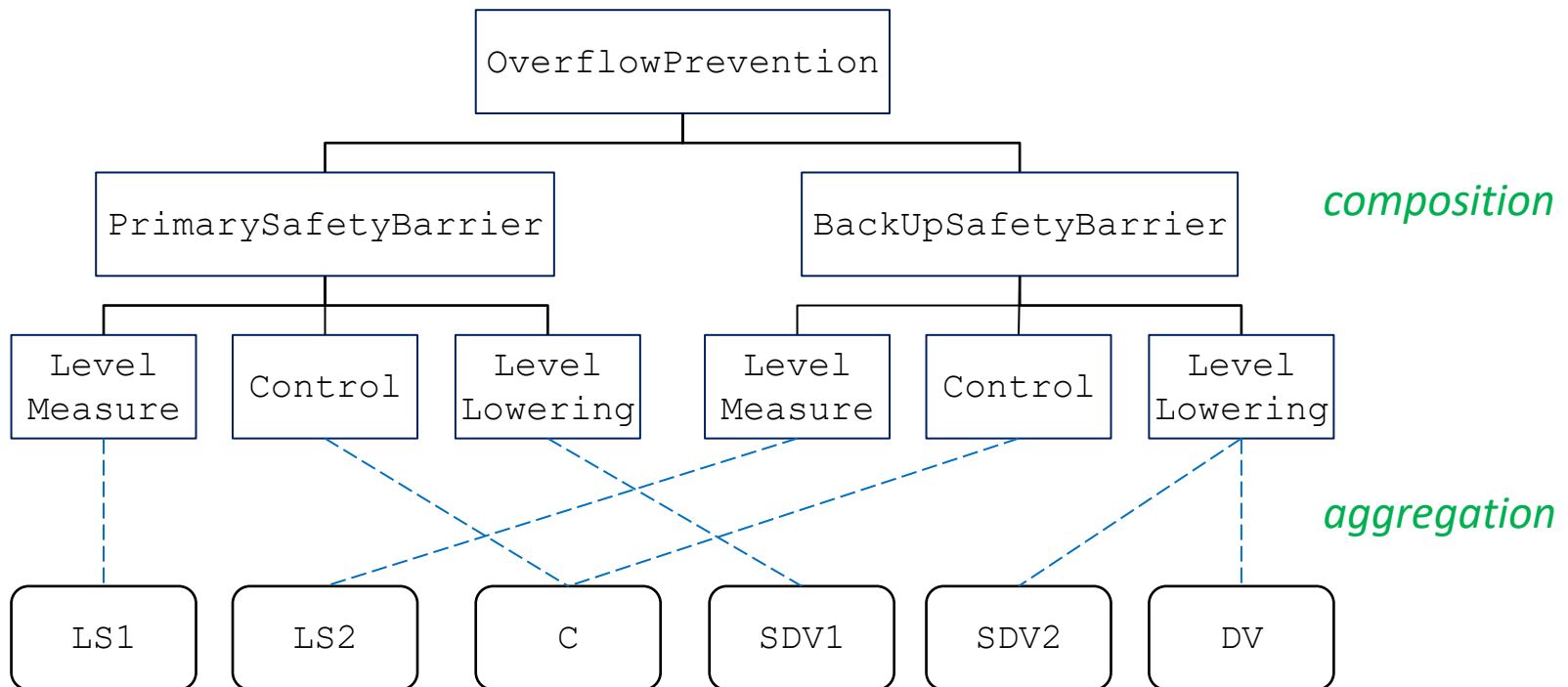
Overflow Protection System

The role of the system is to prevent an overflow of the tank, which may have catastrophic consequences.



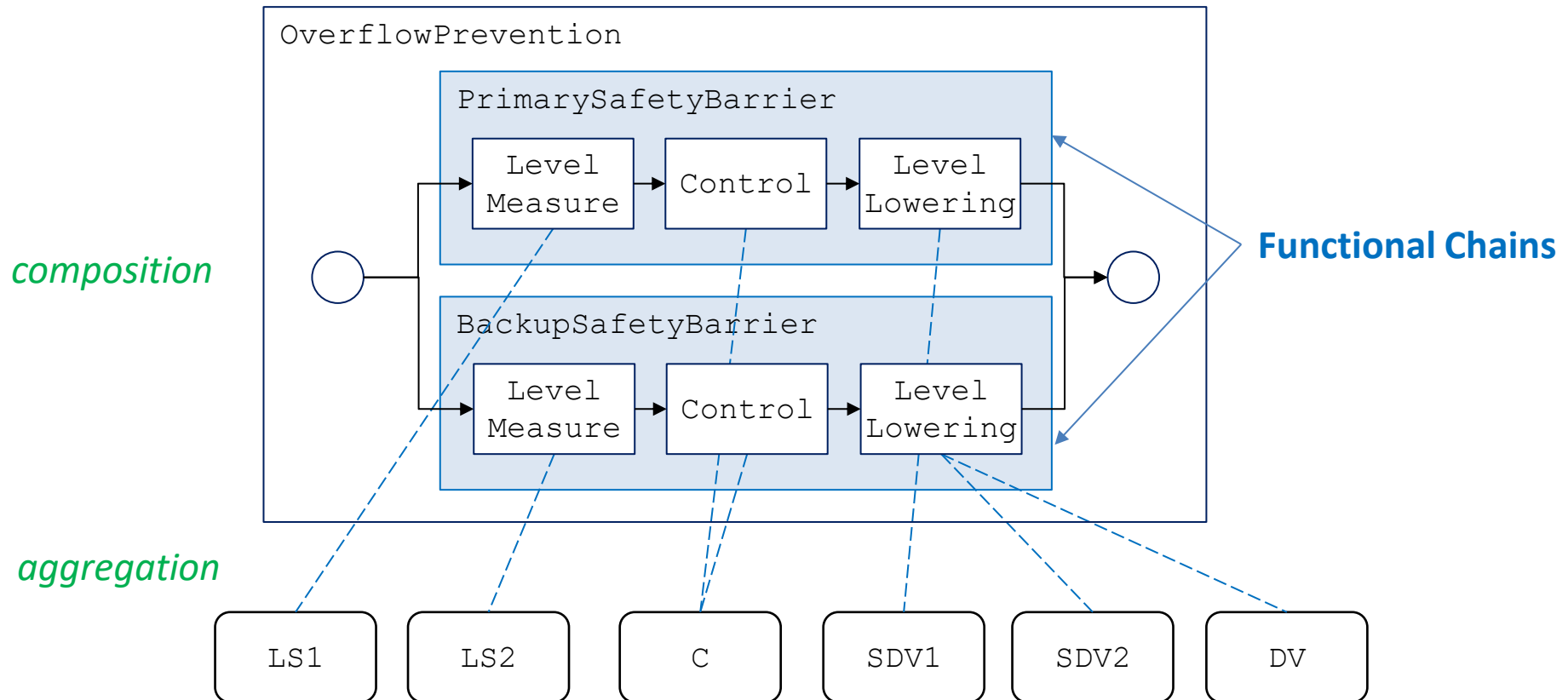
We want to know whether this protection system is good enough for the plant to be operated in sustainable economic and safety conditions, and possibly to suggest modifications to its design.

Logical Architecture



- + Hierarchical structure
- + Functions/Capabilities
- Chaining of function / Flows of information

Functional Chains



- ± Hierarchical structure
- + Functions/Capabilities
- + Chaining of function / Flows of information

Large models are often authored/understood via functional chains

Generic Physical Units

```
domain UnitState {STANDBY, OPERATION, FAILED, MAINTENANCE}
```

```
class PhysicalUnit
```

```
  parameter float fit(unit = "FIT") = 1000;
```

```
  parameter float meanTimeToRepair(unit = "h") = 10;
```

```
  UnitState state(init = STANDBY);
```

```
  port bool operating = state==OPERATION;
```

```
end
```

```
activity PhysicalUnit.Operation
```

```
  trigger: return state==STANDBY;
```

```
  start: state = OPERATION;
```

```
  completion: state = FAILED;
```

```
  duration: return exponentialDeviate(fit * 1.0e-9);
```

```
end
```

```
activity PhysicalUnit.Maintenance
```

```
  trigger: return state==FAILED;
```

```
  start: state = MAINTENANCE;
```

```
  completion: state = STANDBY;
```

```
  duration: return exponentialDeviate(1/meanTimeToRepair);
```

```
end
```

*FIT: number of failures
per 10⁹ hours*

Specific Physical Units

```
class LevelSensor
  extends PhysicalUnit;
  parameter float fit(unit = "FIT") = 1000;
  parameter float meanTimeToRepair(unit = "h") = 24;
end

class Controller
  extends PhysicalUnit;
  parameter float fit(unit = "FIT") = 200;
  parameter float meanTimeToRepair(unit = "h") = 8;
end

class Valve
  extends PhysicalUnit;
  parameter float fit(unit = "FIT") = 2500;
  parameter float meanTimeToRepair(unit = "h") = 48;
end
```

Functional Units

```
class FunctionalUnit
  port bool operational = false;
  port bool in = false;
  port bool out = in and operational;
end
```

Overflow Protection System

```
system OverflowProtectionSystem
  system SIS
    LevelSensor LS1;
    ...
    Valve DV;
  end
  system OverflowProtectionCapability
    system PrimarySafetyBarrier
      ...
    end
    system BackupSafetyBarrier
      ...
    end
    port bool in = true;
    port bool PrimarySafetyBarrier.in = in;
    port bool BackupSafetyBarrier.in = in;
    port bool out = PrimarySafetyBarrier.out or BackupSafetyBarrier.out;
  end
  port bool available = OverflowProtectionCapability.out;
end
```

OverflowProtectionSystem



Safety Barrier

```
system OverflowProtectionSystem
  system SIS ... end
  system OverflowProtectionCapability
    system PrimarySafetyBarrier
      port bool in = false;
      FunctionalUnit LevelMeasure;
      FunctionalUnit Control;
      FunctionalUnit LevelLowering;
      port bool LevelMeasure.operational = main.SIS.LS1.operating;
      port bool LevelMeasure.in = in;
      port bool Control.operational = main.SIS.C.operating;
      port bool Control.in = LevelMeasure.out;
      port bool LevelLowering.operational = main.SIS.SDV1.operating;
      port bool LevelLowering.in = Control.out;
      port bool out = LevelLowering.out;
    end
    system BackupSafetyBarrier ... end
  ...
end
...
end
```

- *Functional units aggregate the states of physical units.*
- *The functional chain is described by "plugging" functional units.*