Safety Assessment of an Electrical System with AltaRica 3.0

Hala MORTADA (LIX, Ecole Polytechnique)

Tatiana PROSVIRNOVA (LIX, Ecole Polytechnique)

Antoine RAUZY (LGI, Ecole Centrale)

Outline

- Case study: an Electrical System
- Structural constructs
- Behavior modeling
  - —Block diagrams
  - —Common cause failures
  - —Repairs, on demand failures and reconfigurations
  - —Shared resources

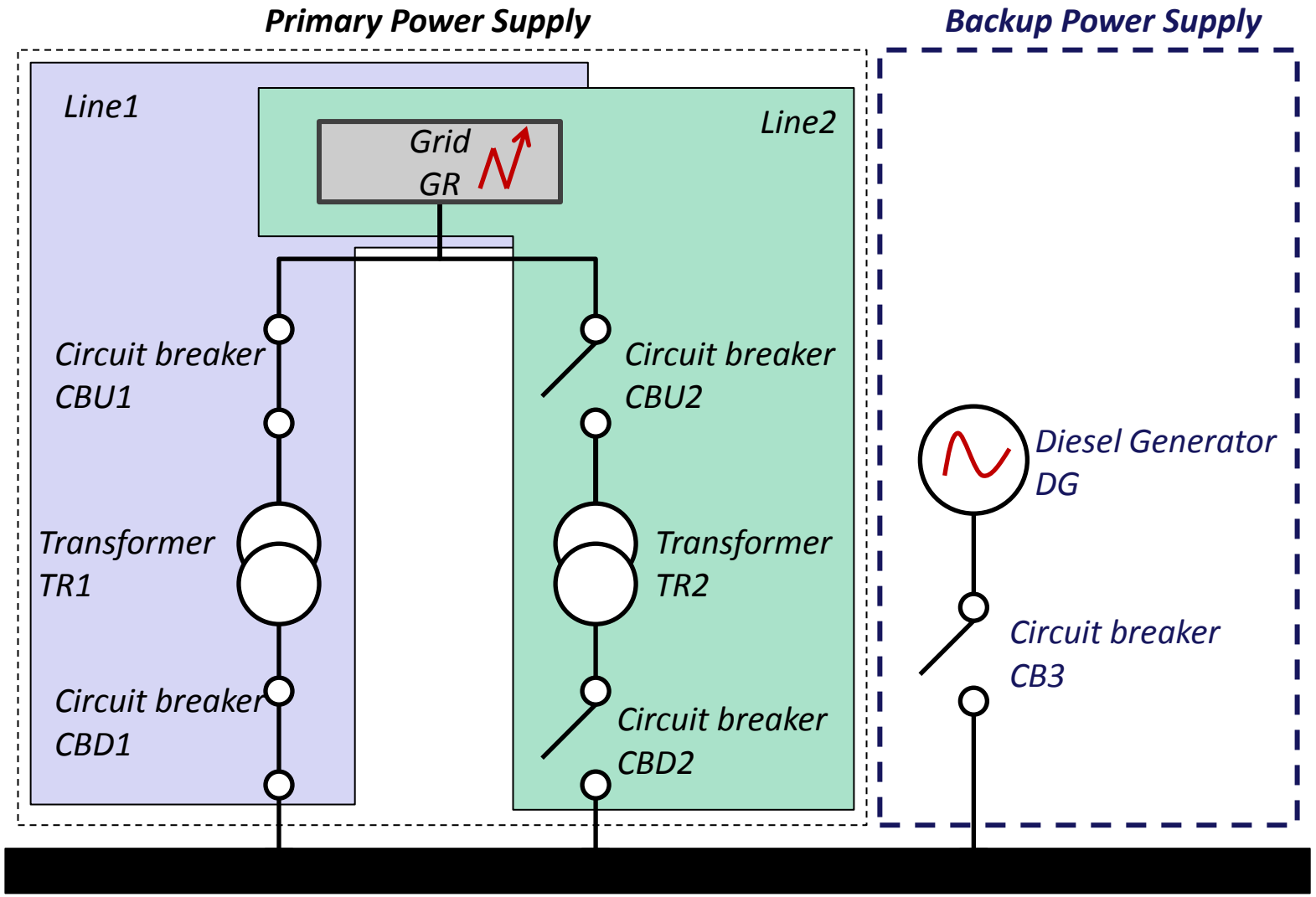
Starting from the same root model, different variants can be obtained by successive refinements.

Outline

- **Case study: an Electrical System**
- Structural constructs
- Behavior modeling
  - —Block diagrams
  - —Common cause failures
  - —Repairs, on demand failures and reconfigurations
  - —Shared resources

Starting from the same root model, different variants can be obtained by successive refinements.

Case study: an Electrical System



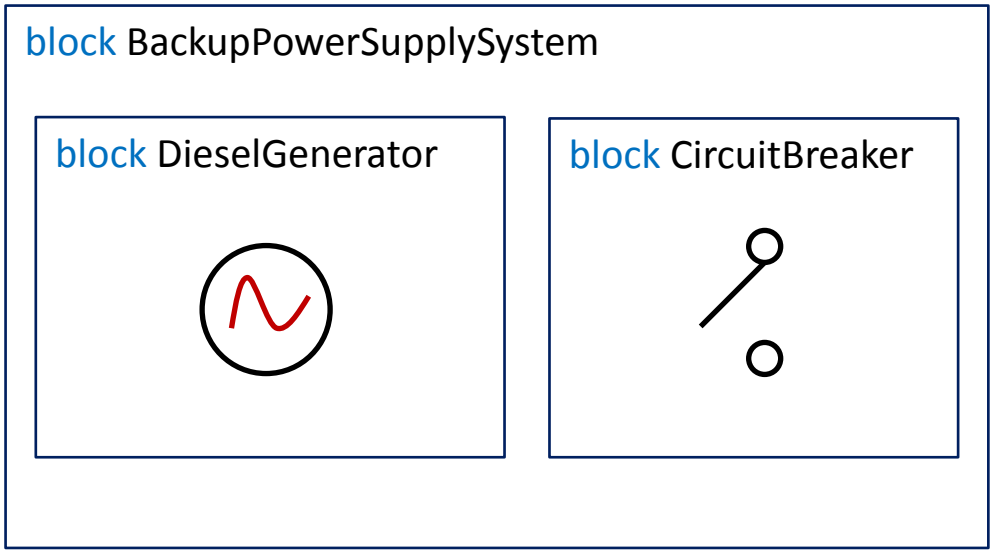
Outline

- Case study: an Electrical System
- **Structural constructs**
- Behavior modeling
  - —Block diagrams
  - —Common cause failures
  - —Repairs, on demand failures and reconfigurations
  - —Shared resources

Starting from the same root model, different variants can be obtained by successive refinements.

Blocks

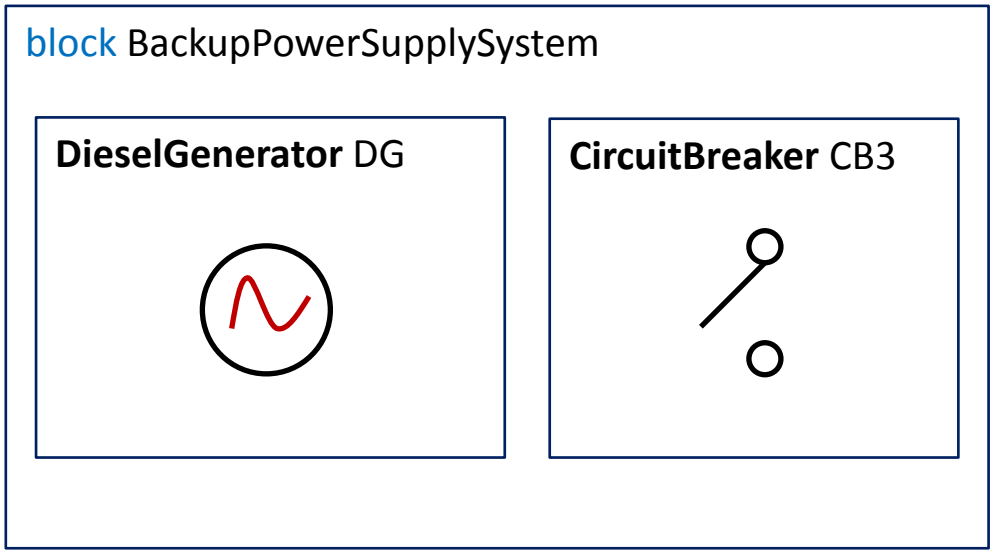
- ***Blocks*** *represent components having a* ***unique occurrence*** *in the model*
- *They are the basic construct to group model elements*
- *A model is thus a* ***hierarchy*** *of nested blocks*
- ***Blocks*** *come from* ***prototype-oriented languages***




```
block BackupPowerSupplySystem

    block DieselGenerator
        // elements of the diesel generator
    end

    block CircuitBreaker
        // elements of the circuit breaker
    end

     /* elements to connect
        the diesel generator and the circuit
     breaker*/
end
```

Classes

- *A **class** is a separately defined, reusable ("on-the-shelf") **block***
- *A **block** can aggregate (contain) other **blocks** as well as instances, i.e. copies, of classes*
- *A **class** can aggregate **blocks** and instances of **classes***
- *Definitions of **classes** cannot be recursive nor circular*
- ***Classes** come from **object-oriented modeling languages***




```
class DieselGenerator
      // elements of the diesel generator
end
class CircuitBreaker
     // elements of the circuit breaker
end
block BackupPowerSupplySystem
      DiseselGenerator DG;
      CircuitBreaker CB3;
      /* elements to connect
         the diesel generator DG and the
        circuit breaker CB3*/
end
```

Flattening

- *The **semantics** of AltaRica 3.0 is defined by means of **flattening rules**, i.e. that the hierarchy of blocks and instances of classes is collapsed into a single block*
- ***Flattening rules** depend on the type of hierarchical links:*
  - *—Aggregation*
  - *—Inheritance*
  - *—Reference (“embeds” clause)*

Aggregation

- *A hierarchical link: the **is-a-part-of** link*
  - *e.g. the left-front wheel is a part of the car*
- *In AltaRica 3.0, a **block** or a **class** can **aggregate blocks** and instances of **classes***
- *Flattening rules:*
  - *Elements of the nested block (or the instance of class) are copied;*
  - *Names of the elements are prefixed with the name of the block (or the instance of class)*

```
class  DieselGenerator
      Boolean working (init = true);
      event failure;
      transition
            failure : working -> working :=false;
end
block BackupPowerSupplySystem
      DieselGenerator  DG;
end
```

flattening

```
block BackupPowerSupplySystem
      Boolean DG.working (init = true);
      event DG.failure;
      transition
        DG.failure : DG.working -> DG.working :=false;
end
```

Inheritance

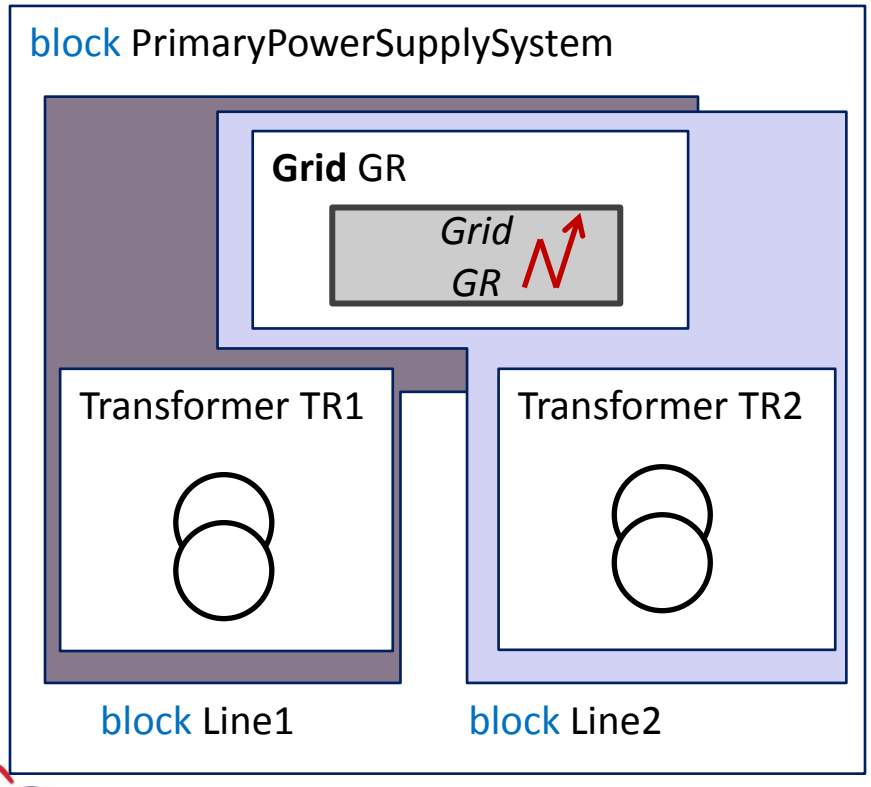
- *A hierarchical link: the **is-a-kind-of** link*
  - *e.g. a car is a kind of vehicle*
- *In AltaRica 3.0, a **block** or a **class** can **inherit** from another **class**.*
- *Flattening rules for **inheritance** are the same as those for **aggregation**, with the exception that the elements of the inherited components are copied without any prefix*

```
class NonRepairableComponent
        Boolean working (init = true);
        event failure;
        transition
                failure : working -> working :=false;
end

class Pump extends NonRepairableComponent
        Boolean  inflow (reset = false);
        Boolean outflow (reset = false);
        assertion
                outflow := inflow and working;
end
```


```
class Pump
        Boolean working (init = true);
        Boolean  inflow (reset = false);
        Boolean outflow (reset = false);
        event failure;
        transition
                failure : working -> working :=false;
        assertion
                outflow := inflow and working;
end
```

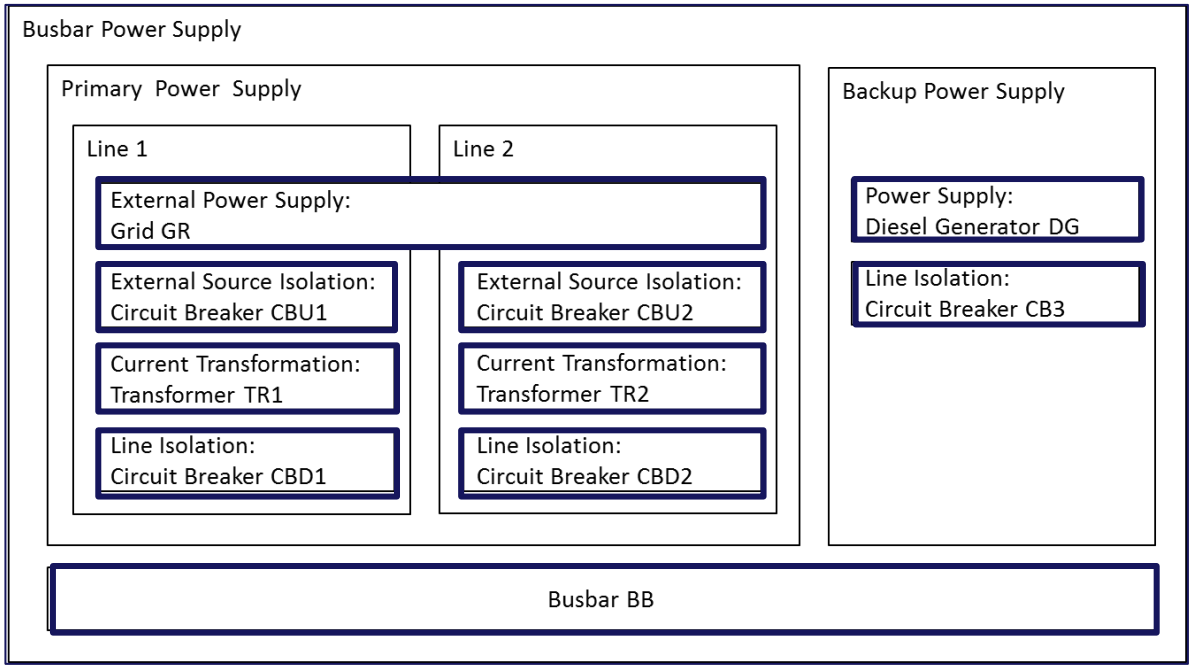
Reference: the "embeds" clause

- *Hierarchical decompositions are often thought only as tree-like structures. But in practice it is often convenient for two branches to share a component (especially when reasoning in functional terms).*
- ***AltaRica 3.0*** *makes it possible to* ***share blocks*** *and* ***instances of classes*** *through the "****embeds****" clause.*




```
block PrimaryPowerSupplySystem
      Grid GR;

      block Line1
            embeds GR;
            Transformer TR1;
      end

      block Line2
            embeds GR;
             Transformer TR2;
      end
end
```

System architecture



Class instances

Blocks

```
block ElectricalSystem
 block PrimaryPowerSupplySystem
  Grid GR;
  block Line1
   embeds GR;
   CircuitBreaker CBU1, CBD1;
   Transformer TR1;
  end
  block Line2
   embeds GR;
   CircuitBreaker CBU2, CBD2;
   Transformer TR2;
  end
 end
 block BackupPowerSupplySystem
  DieselGenerator DG;
  CircuitBreaker CB3;
 end
  Busbar BB;
end
```

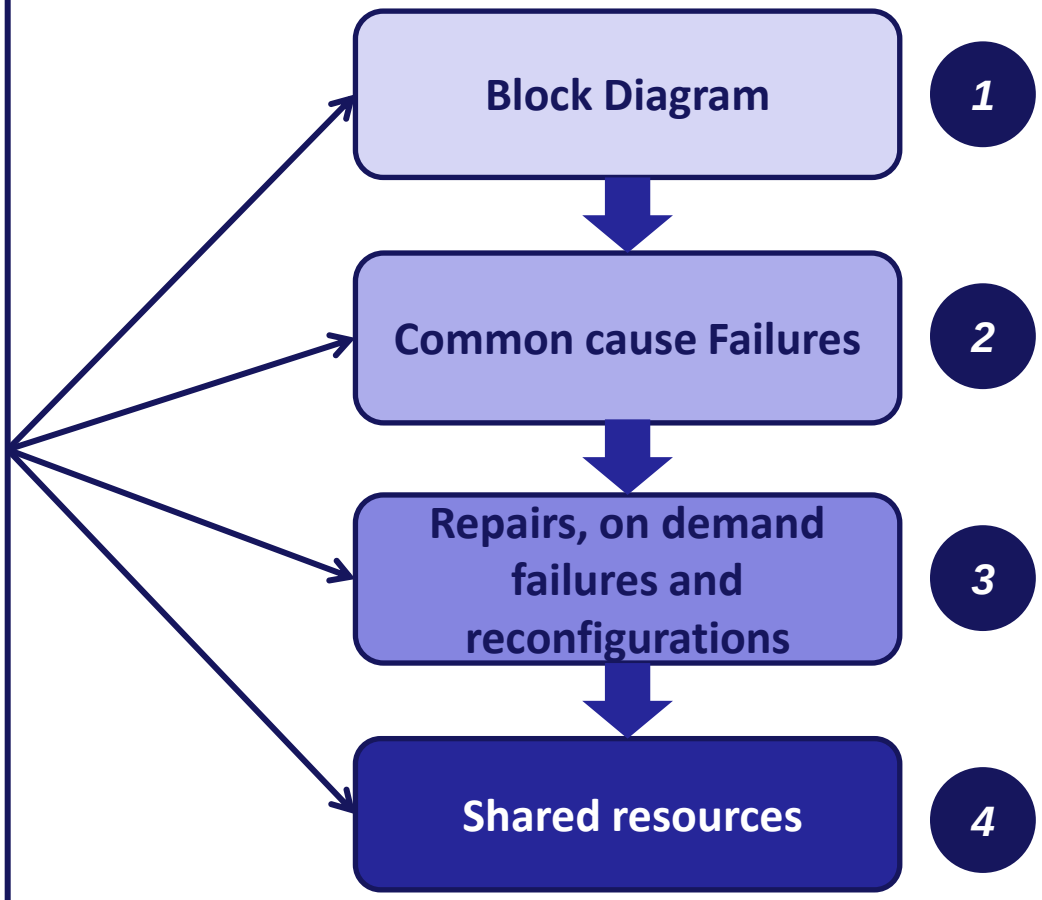
Outline

- Case study: an Electrical System
- Structural constructs
- **Behavior modeling**
  - —Block diagrams
  - —Common cause failures
  - —Repairs, on demand failures and reconfigurations
  - —Shared resources

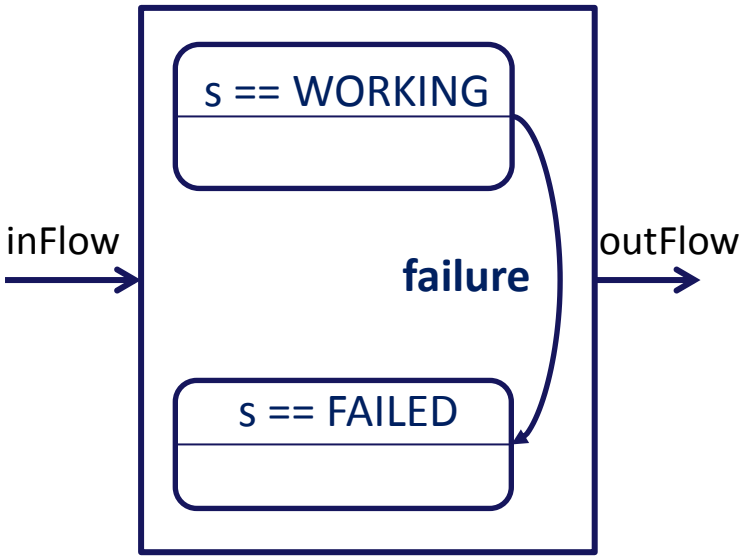
Starting from the same root model, different variants can be obtained by successive refinements.

Behavior modeling

```
block ElectricalSystem
 block PrimaryPowerSupplySystem
  Grid GR;
  block Line1
   embeds GR;
   CircuitBreaker CBU1, CBD1;
   Transformer TR1;
  end
  block Line2
   embeds GR;
   CircuitBreaker CBU2, CBD2;
   Transformer TR2;
  end
 end
 block BackupPowerSupplySystem
  DieselGenerator DG;
  CircuitBreaker CB3;
 end
 Busbar BB;
end
```



Block diagram: modeling

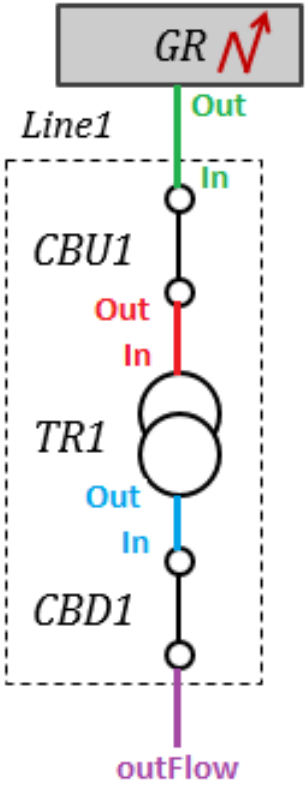


All the components are represented by the class NonRepairableComponent with input and output flows

```
domain ComponentState { WORKING, FAILED }

class NonRepairableComponent
  ComponentState s (init = WORKING);
  event failure (delay = exponential(lambda));
  parameter Real lambda = 0.001;
  transition
    failure: s == WORKING  -> s := FAILED;
end

class Transformer extends NonRepairableComponent;
  Boolean inFlow, outFlow (reset = FALSE);
  assertion
    outFlow := (s == WORKING) and inFlow;
end
```

Block diagram: modeling


```
block ElectricalSystem
  Boolean outFlow (reset = false);
  block PrimaryPowerSupply
    …
    block Line1
      …
      assertion
        CBU1.inFlow := GR.outFlow;
        TR1.inFlow := CBU1.outFlow;
        CBD1.inFlow := TR1.outFlow;
    end
    block Line2
      ... // similar to Line1
    end
    assertion
      outFlow := Line1.outFlow or Line2.outFlow;
  end
  ...
  assertion
    outFlow := PrimaryPowerSupply.outFlow or BackupPowerSupply.outFlow;
end
```



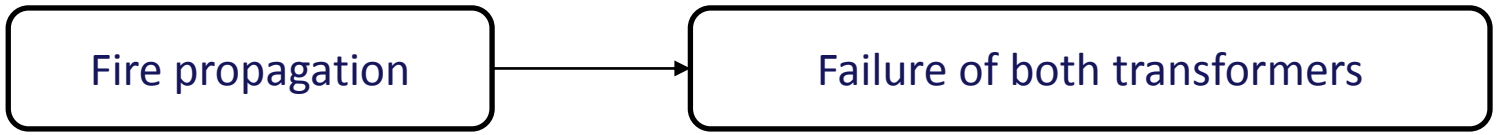
Block diagram: assessment

- *Static model*
- *Is compiled into **Fault Trees***
- *Calculated Minimal cutsets (MCS):*

MCS	MCS
GR.failure, DG.failure	GR.failure, CB3.failure
CBU1.failure, CBU2.failure, DG.failure	CBU1.failure, CBU2.failure, CB3.failure
CBU1.failure, TR2.failure, DG.failure	CBU1.failure, TR2.failure, CB3.failure
CBU1.failure, CBD2.failure, DG.failure	CBU1.failure, CBD2.failure, CB3.failure
TR1.failure, CBU2.failure, DG.failure	TR1.failure, CBU2.failure, CB3.failure
TR1.failure, TR2.failure, DG.failure	TR1.failure, TR2.failure, CB3.failure
TR1.failure, CBD2.failure, DG.failure	TR1.failure, CBD2.failure, CB3.failure
CBD1.failure, CBU2.failure, DG.failure	CBD1.failure, CBU2.failure, CB3.failure
CBD1.failure, TR2.failure, DG.failure	CBD1.failure, TR2.failure, CB3.failure
CBD1.failure, CBD2.failure, DG.failure	CBD1.failure, CBD2.failure, CB3.failure

Common cause failures

- *Consider that the two **transformers** have a **common cause failure** due to a fire propagation.*
- *Assume that the CCF failure rate **lambdaCCF =1.0e-5**.*



Common cause failures: modeling


```
block PowerSupplySystem
  block PrimaryPowerSupplySystem
    Grid GR;
    block Line1
      …
    end
    block Line2
      …
    end
    event TransformersCCF (delay = exponential (lambdaCCF));
    parameter Real lambdaCCF = 1.0e–5;
    transition
      TransformersCCF: ?Line1.TR1.failure & ?Line2.TR2.failure;
    …
  end
…
end
```

Common cause failures: assessment

- *Static model*
- *Is compiled into **Fault Trees***
- *Two additional minimal cutsets:*

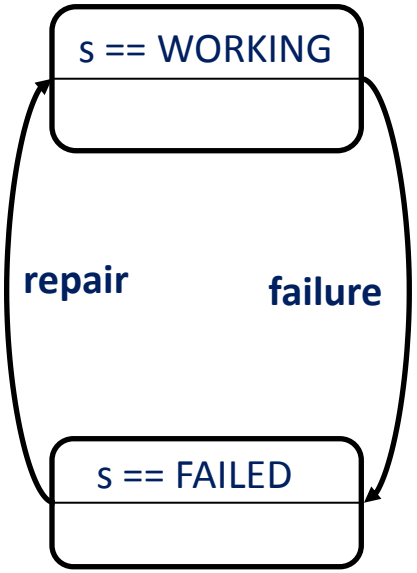
MCS
TransformersCCF, DG.failure
TransformersCCF, CB3.failure

Repairs, on demand failures and reconfigurations

- *Consider that*
  - *The **circuit breakers** can be open, closed and can have **on demand failures** (failure to open or failure to close) with a given probability **gamma = 10^{-3}.***
  - *The **diesel generator** can be started or stopped and can have **on demand failure** (failure to start).*
  - *In the **initial state** the power is transmitted to the Busbar via the Line 1, i.e. the **circuit breakers** CBU1 and CBD1 are closed, the **circuit breakers** CBU2 and CBD2 are open, the **diesel generator** DG is stopped, the **circuit breaker** CB3 is open.*
  - *If the Line 1 fails, the system should be **reconfigured** to use the Line 2 (i.e. the **circuit breakers** CBU2 and CBD2 should be closed). If both lines are failed, the **diesel generator** DG is attempted to start and the **circuit breaker** CB3 is attempted to close.*

Repairs, on demand failures and reconfigurations: modeling

How to model a repairable component?

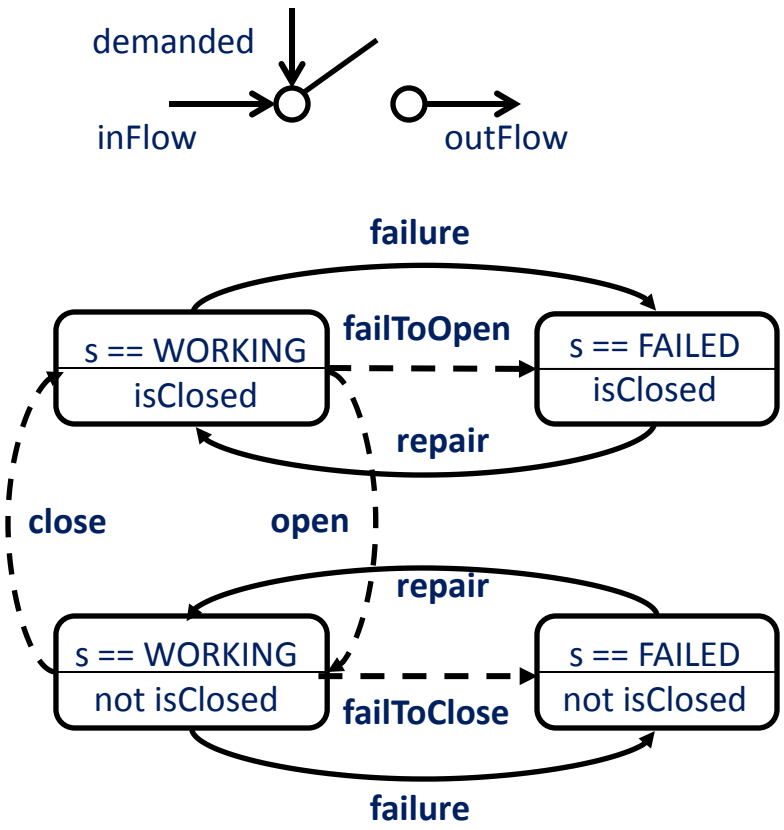



```
class RepairableComponent
  extends NonRepairableComponent;
  Boolean failed (reset = false);
  event repair (delay = exponential(mu));
  parameter Real mu = 0.1;
  transition
     repair: s == FAILED -> s := WORKING;
  assertion
    failed := s == WORKING;
end

class Grid extends RepairableComponent;
  Boolean outFlow (reset = false);
  assertion
    outFlow := s==WORKING;
end
```

The grid GR and the transformers TR1 and TR2 are repairable components

Repairs, on demand failures and reconfigurations: modeling

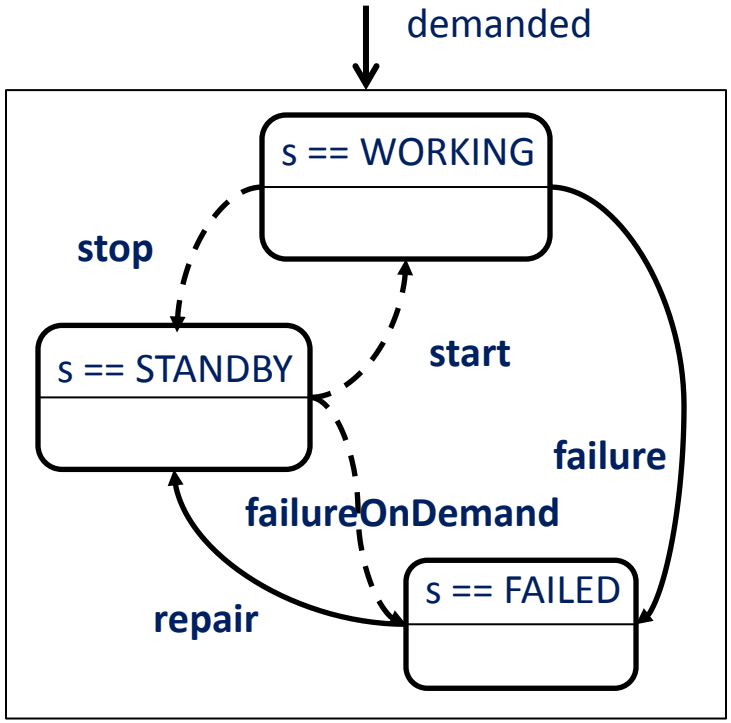
How to model a circuit breaker?




```
class CircuitBreaker extends RepairableComponent;
  Boolean isClosed (init = true);
  Boolean  demanded (reset = false);
event failToOpen (delay = 0, expectation = gamma1);
event failToClose (delay = 0, expectation = gamma2);
event open (delay = 0, expectation = 1 - gamma1);
event close (delay = 0, expectation = 1 - gamma2);
parameter Real gamma1 = 0.001;
parameter Real gamma2 = 0.001;

 transition
   failToOpen:  isClosed and (s == WORKING) and  not demanded->
                                        s := FAILED;
   open: isClosed and (s == WORKING) and  not demanded->
                                      isClosed := false;
   failToClose: not isClosed and (s == WORKING) and demanded->
                                        s := FAILED;
   close:  not isClosed and (s == WORKING) and demanded->
                                      isClosed := true;
assertion
   outFlow := isClosed and s == WORKING and inFlow;
end
```

Repairs, on demand failures and reconfigurations: modeling

How to model a spare component?

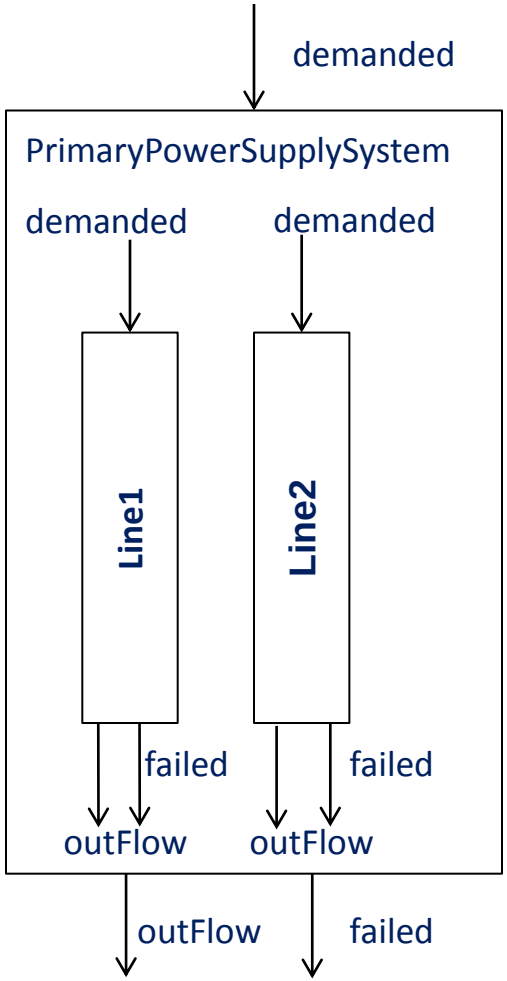


The diesel generator DG is represented by a spare component

```
domain SpareComponentState {STANDBY, WORKING, FAILED}
class SpareComponent;
  SpareComponentState s  (init = STANDBY);
  Boolean failed (reset = false);
  Boolean demanded (reset = FALSE);
  parameter Real lambda = 1.0e-4;
  parameter Real mu = 0.1;
  parameter Real gamma = 0.001;
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  event stop (delay = 0);
  event start (delay = 0, expectation = 1-gamma);
  event failureOnDemand (delay = 0, expectation = gamma);
  event repair (delay = exponential(mu));
transition
    stop: s == WORKING and not demanded -> s := STANDBY;
    start: s == STANDBY and demanded -> s := WORKING;
    failureOnDemand: s == STANDBY and demanded ->
                                    s := FAILED;
    repair: s == FAILED -> s := STANDBY;
    failure: s == WORKING -> s := FAILED;
assertion
    failed := s == FAILED.
end
```

Repairs, on demand failures and reconfigurations: modeling

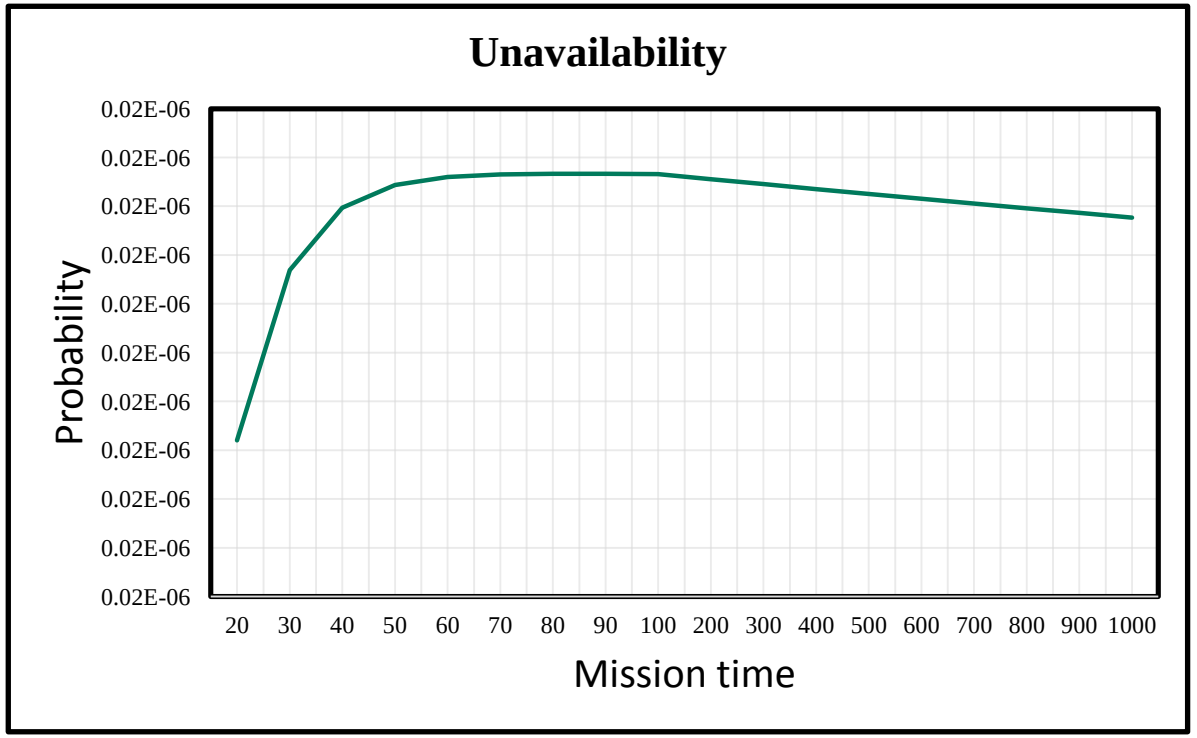
How to represent reconfigurations?




```
block PowerSupplySystem
  block PrimaryPowerSupplySystem
    Boolean failed, demanded (reset = false);
    Grid GR;
    block Line1
      Boolean failed, demanded (reset = false);
      …
      assertion
        CBU1.demanded := demanded;
        CBD1.demanded := demanded;
        failed := GR.failed or CBU1.failed or TR1.failed or CBD1.failed;
    end
    …
    assertion
      Line1.demanded := not Line1.failed and not Line2.outFlow;
      Line2.demanded := not Line2.failed and not Line1.outFlow;
  …
end
```

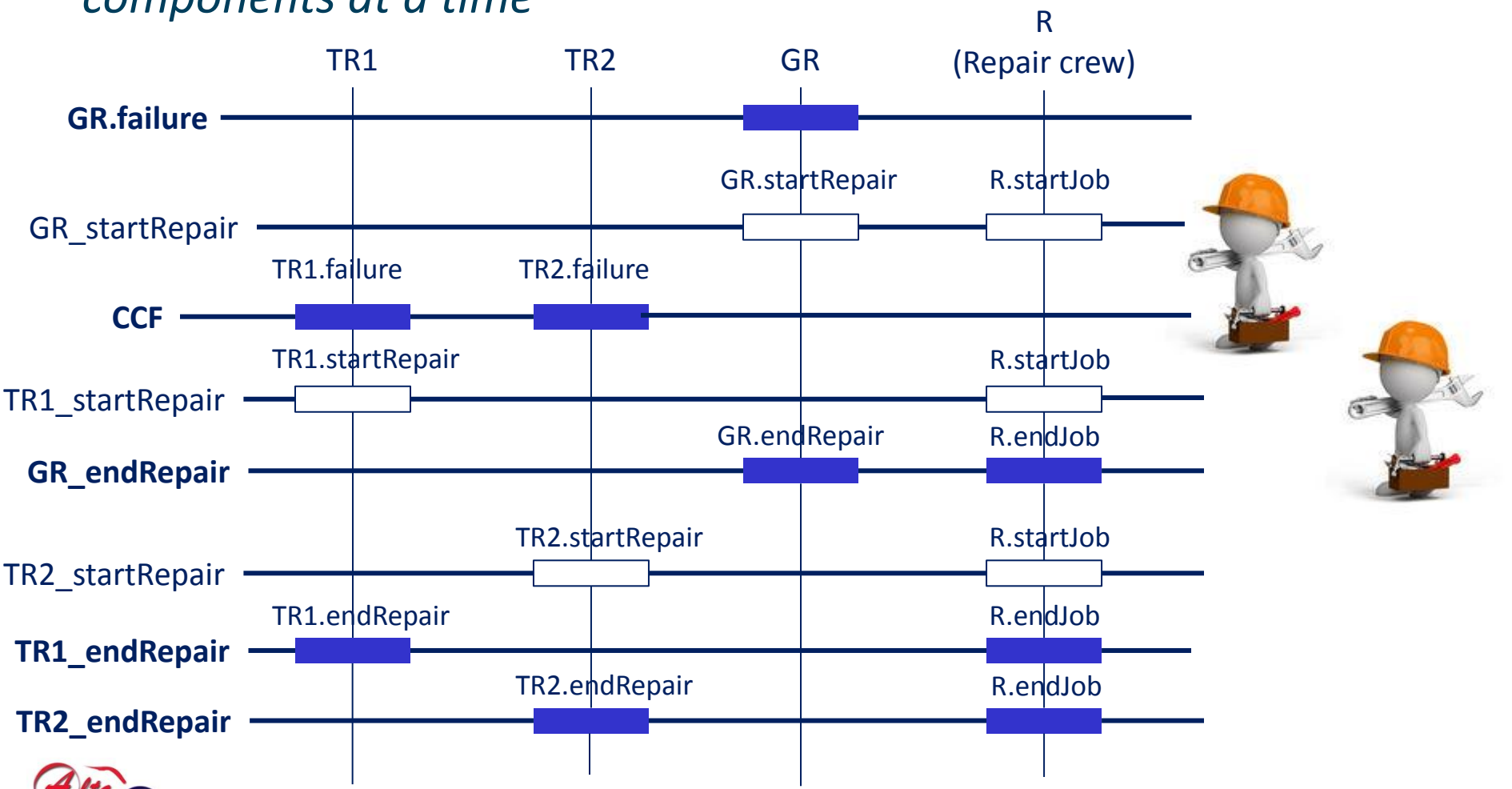
On demand failures and reconfigurations: assessment

- *Dynamic model*
- *Is compiled into **Markov chains***
- *Assessment of the generated Markov chain: unavailability calculation*



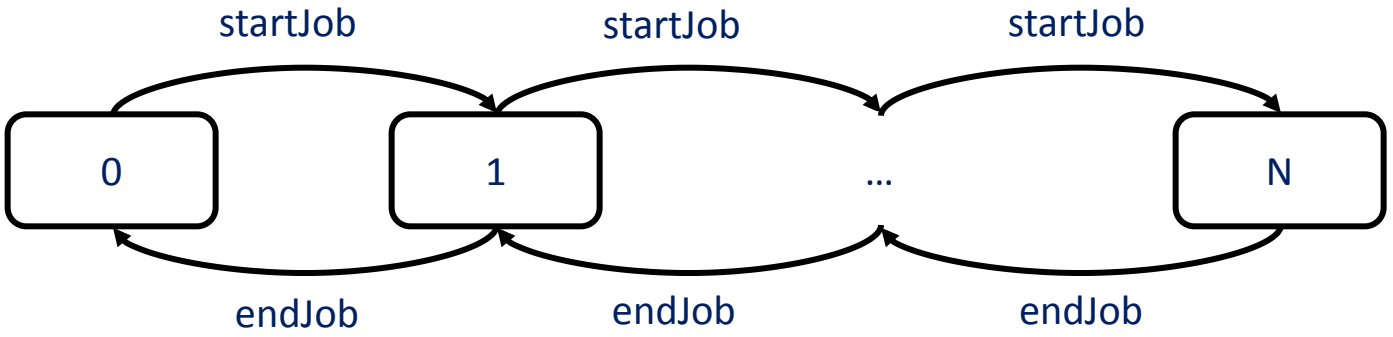
Shared resources

Consider that a limited repair crew can work on at most two components at a time



Shared resources: modeling

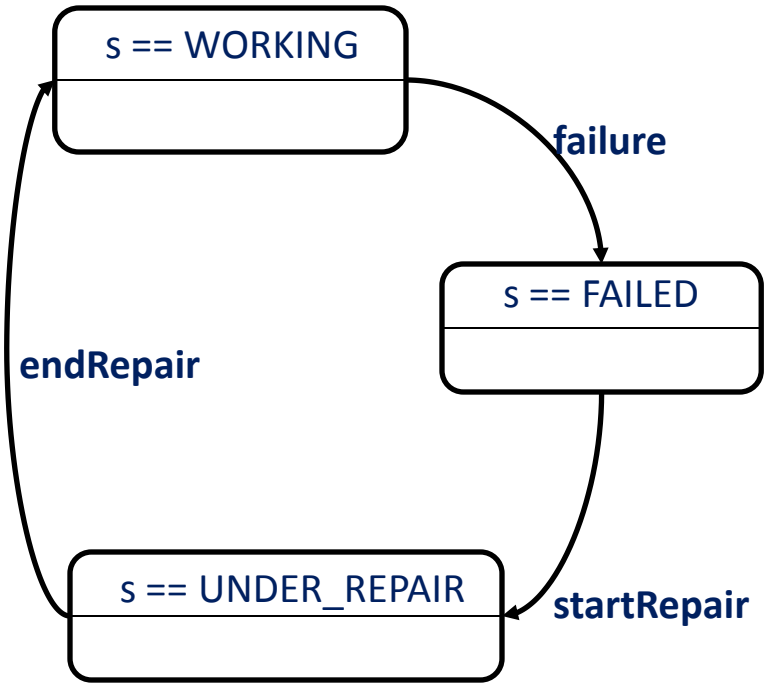
How to model a repair crew?




```
class RepairCrew
  Integer numberOfBusyRepairers (init = 0);
  parameter Integer totalNumberOfRepairers = 2;
  event startJob, endJob;
transition
    startJob: numberOfBusyRepairers < totalNumberOfRepairers ->
                          numberOfBusyRepairers :=   numberOfBusyRepairers + 1;
    endJob: numberOfBusyRepairers > 0 -> numberOfBusyRepairers := numberOfBusyRepairers - 1;
end
```

Shared resources: modeling

How to model a repairable component?




```
domain RepairableComponentState {WORKING, FAILED,
UNDER_REPAIR}

class RepairableComponent
  RepairableComponentState s (init = WORKING);
  Boolean failed (reset = false);
  event failure (delay = exponential(lambda));
  event startRepair;
  event endRepair (delay = exponential(mu));
  parameter Real mu = 0.;
  parameter Real lambda = 1.0 e-4;
  transition
    failure: s == WORKING -> s := FAILED;
    startRepair: s == FAILED ->  s := UNDER_REPAIR ;
    endRepair: s == UNDER_REPAIR -> s := WORKING;
  assertion
    failed := s == FAILED or s == UNDER_REPAIR;
end
```

Shared resources: modeling

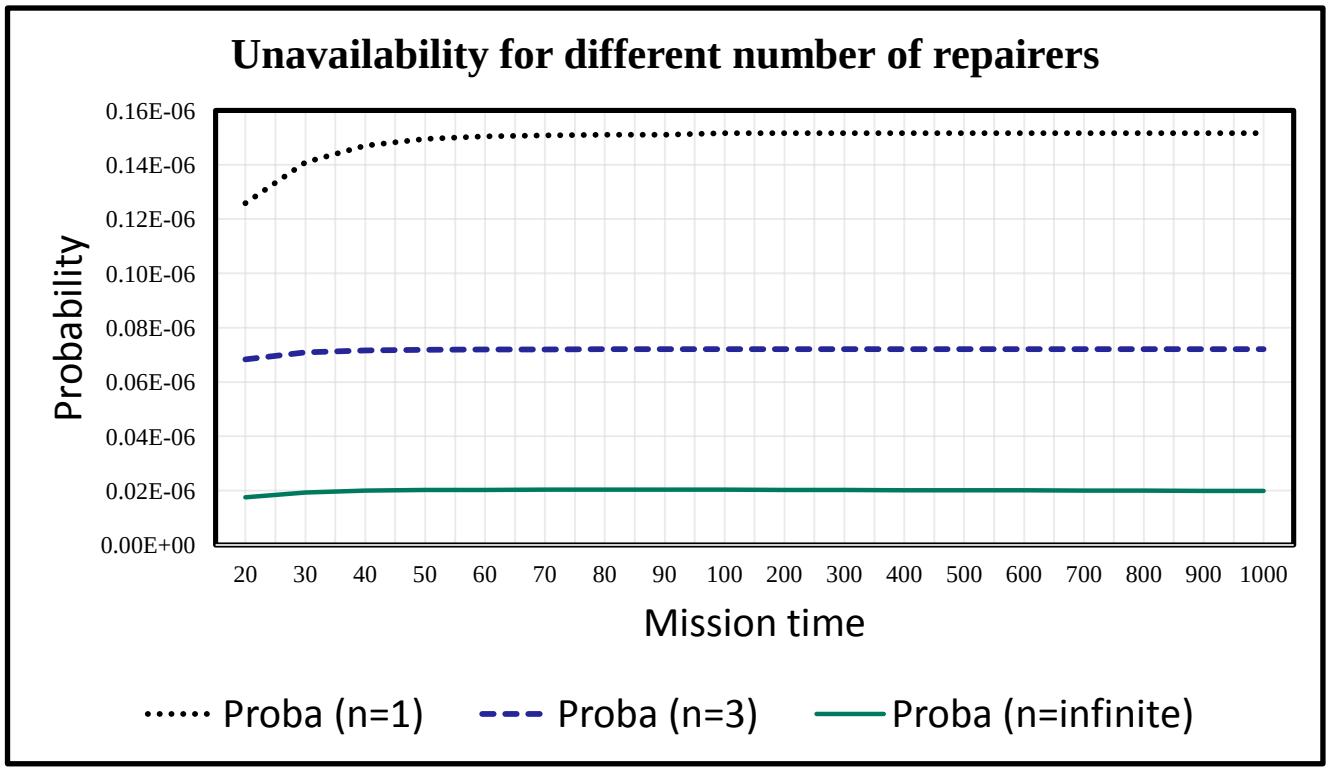
How to synchronize events?

```
block PowerSupplySystem
  block PrimaryPowerSupplySystem
  …
  end
  block BackupPowerSupplySystem
  …
  end
  …
  RepairCrew R(totalNumberOfRepairers = 2);
  event GR_startRepair;
  event GR_endRepair;
  …
  transition
   GR_startRepair: !R.startJob & !PrimaryPowerSupplySystem.GR.startRepair;
   GR_endRepair: !R.endJob & !PrimaryPowerSupplySystem.GR.endRepair;
   hide R.startJob, R.endJob;
   hide PrimaryPowerSupplySystem.GR.startRepair, PrimaryPowerSupplySystem.GR.endRepair;
    …
end
```

Shared resources: assessment

- *Dynamic model*
- *Is compiled into **Markov chains***
- *Assessment of the generated Markov chain: unavailability calculation*

λ	10^{-4}
γ	10^{-3}
μ	10^{-1}



Conclusion

- **Different variants** of models of the case study have been defined
  - —The first series targeting a compilation into Fault Trees
  - —The second series targeting a compilation into Markov chains
- Each of these variants captures a particular aspect of the system under study
- A model is always a **trade-off** between the accuracy of the description and the ability to perform calculations
- **Refining a model** in successive variants is a good way to seek a good trade-off
- The approach made of successive derivation of variants is a solid ground to build a modeling methodology