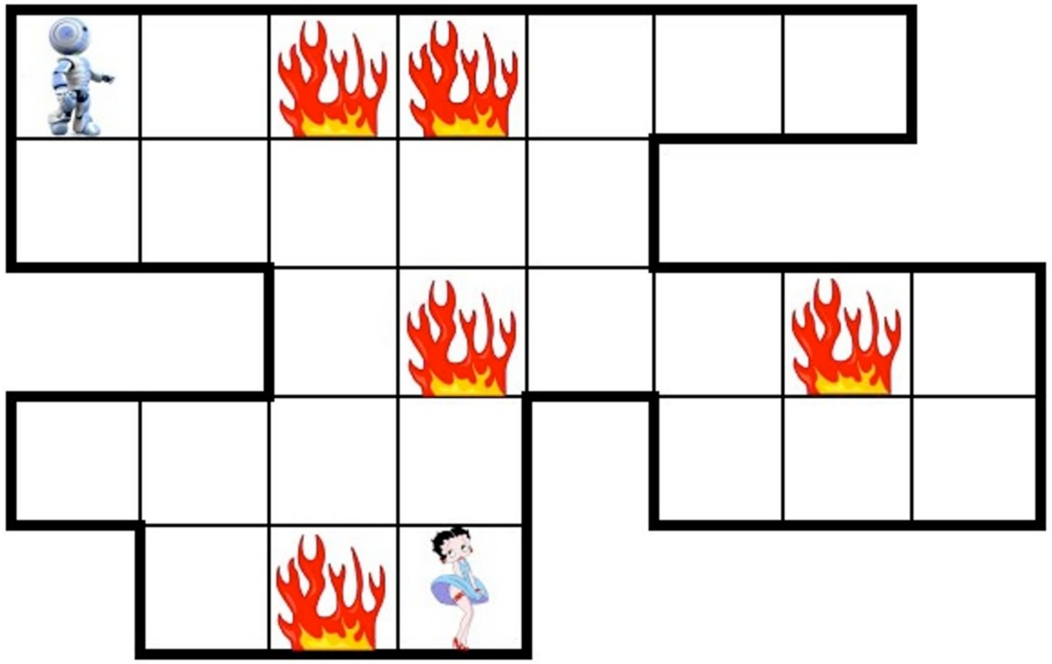
Component-Based Design of Concurrent Software in BIP

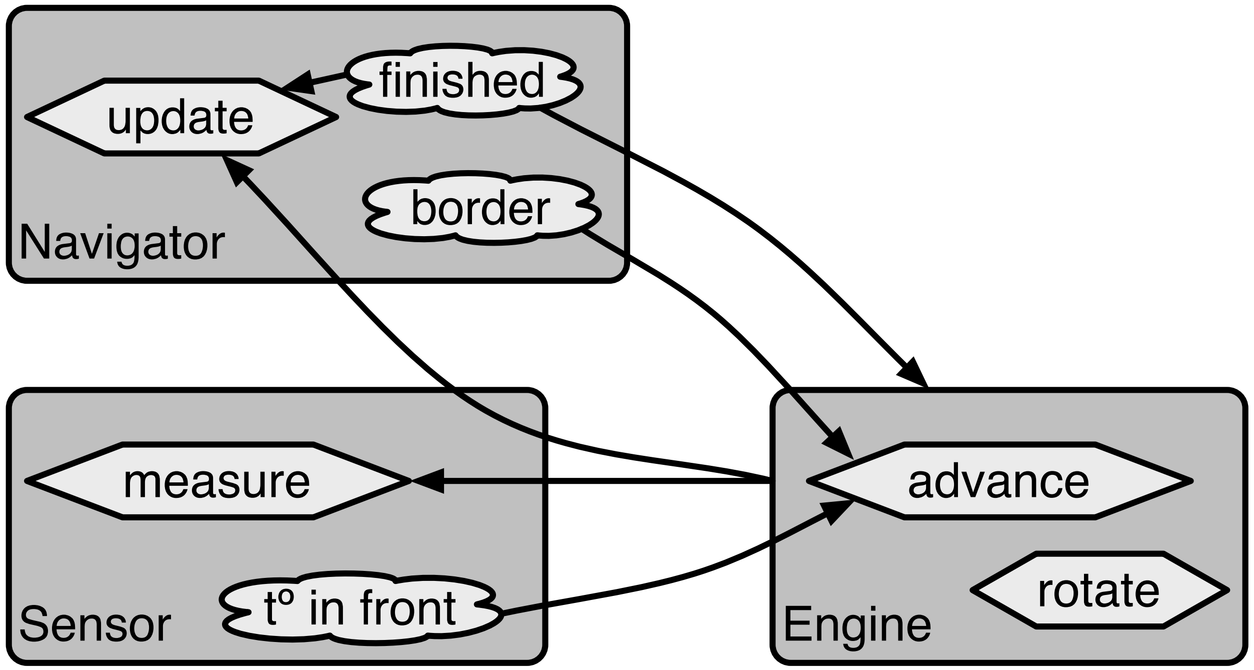
Lecture @ AUTh
16th of October, 2019

Simon Bliudze
https://www.bliudze.me/simon

Inria Lille – Nord Europe

Example: Rescue robot

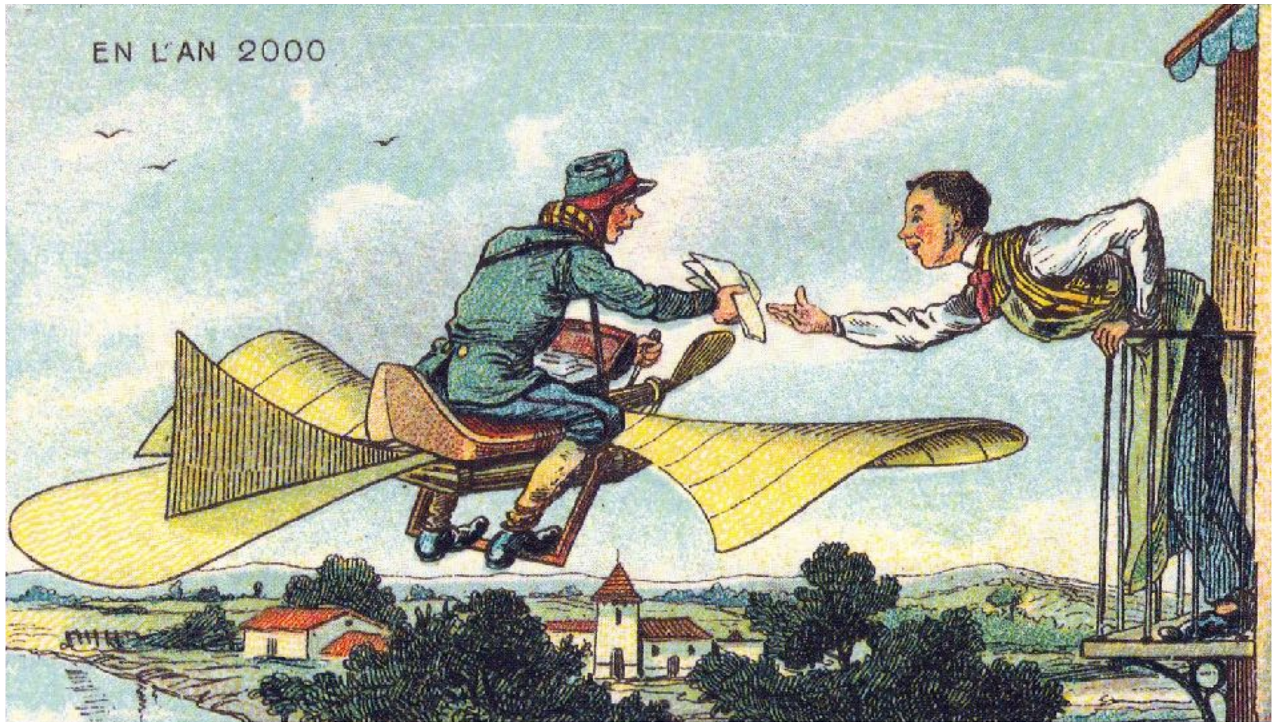




Safety constraints

- Shall not advance and rotate at the same time
- Shall stay within the region
- Shall stay in the area that is safe or hot (but not burning)
- Shall update navigation and sensor data at each move
- When objective is found, the robot shall stop

Coordination



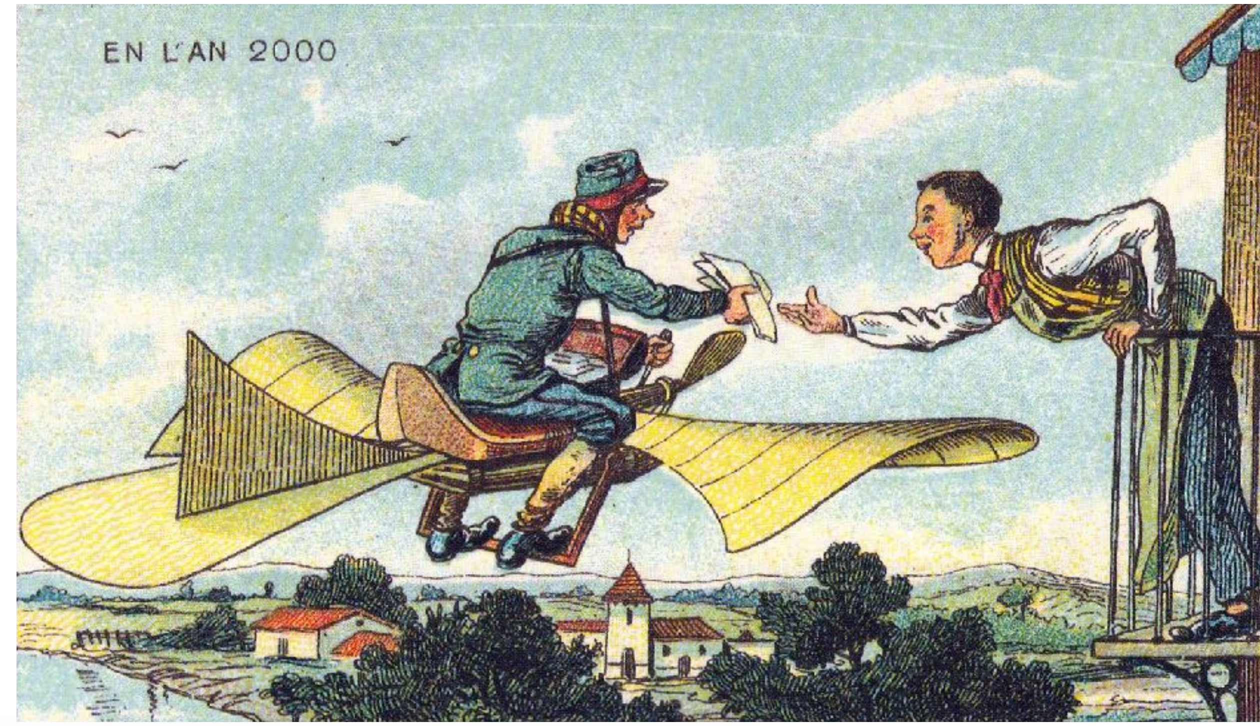
Control-centric

- Synchronisation is primitive
- Locks, semaphores etc.
- Concurrent execution
- Critical systems

Data-centric

- Data exchange is primitive
- Messages, split-join etc.
- Distributed execution
- Data-intensive computation

Coordination



The two views are complementary

Control-centric

- Synchronisation is primitive
- Locks, semaphores etc.
- Concurrent execution
- Critical systems

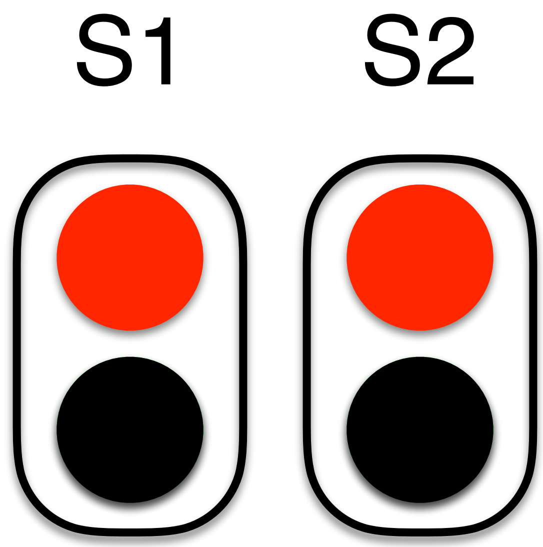
Data-centric

- Data exchange is primitive
- Messages, split-join etc.
- Distributed execution
- Data-intensive computation

Semaphores, locks, monitors, etc.

Coordination based on low-level primitives rapidly becomes unpractical.

Synchronisation



Process 1:

```
...
free(S1);
take(S2);
...
```

Process 2:

```
...
take(S1);
free(S2);
...
```

A simple synchronisation barrier

Synchronisation

Process 1:

```
...
free(S1);
free(S1);
take(S2);
take(S3);
...
```

Process 2:

```
...
take(S1);
free(S2);
free(S2);
take(S3);
...
```

Process 3:

```
...
take(S1);
take(S2);
free(S3);
free(S3);
...
```

Three-way synchronisation barrier

Synchronisation with data transfer

Process 1:

```
x = f1(sh1,sh2);
free(S1);
take(S2);
sh1 = f2(sh1,x);
free(S1);
take(S2);
x = f3(sh1,sh2);
```

Process 2:

```
y = g1(sh1,sh2);
take(S1);
free(S2);
sh2 = g2(y,sh2);
take(S1);
free(S2);
y = g3(sh1,sh2);
```

Coordination mechanisms mix up with computation and do not scale.
Code maintenance is a nightmare!

Synchronisation with data transfer

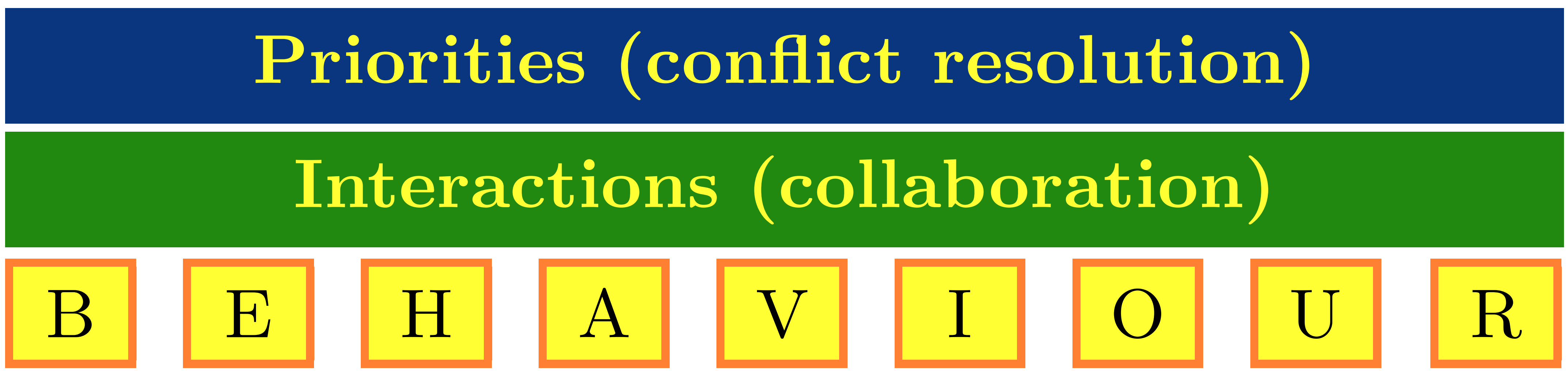
Process 1:

```
x = f1(sh1,sh2);
free(S1);
take(S2);
sh1 = f2(sh1,x);
free(S1);
take(S2);
x = f3(sh1,sh2);
```

Process 2:

```
y = g1(sh1,sh2);
take(S1);
free(S2);
sh2 = g2(y,sh2);
take(S1);
free(S2);
y = g3(sh1,sh2);
```

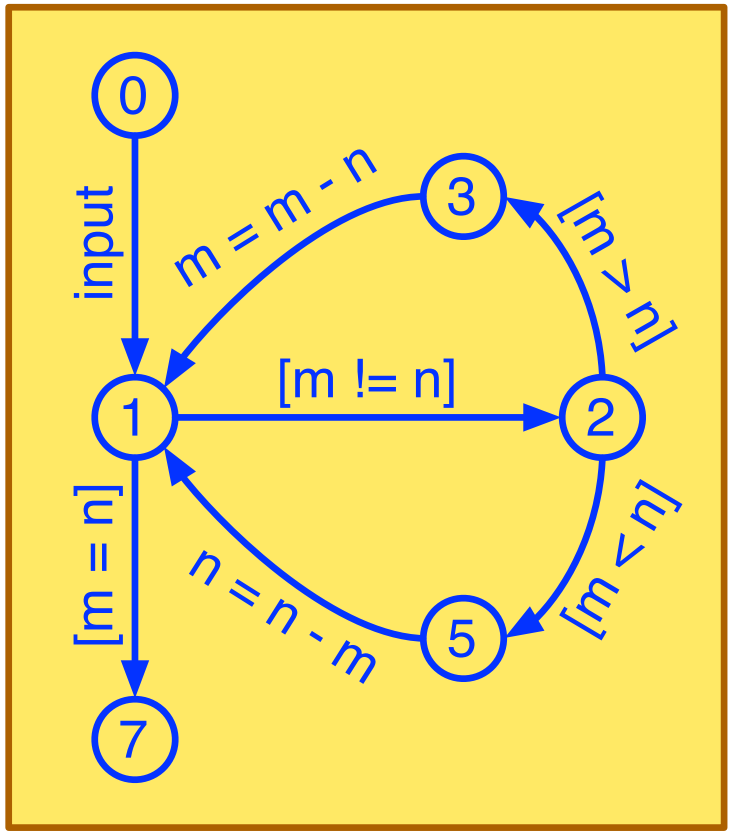
Coordination mechanisms mix up with computation and do not scale.
Code maintenance is a nightmare!



The BIP framework

Components

```
input(m,n>0);
while(m != n){
 if (m > n)
   m = m - n;
 else //m < n
   n = n - m;
}
//m=n=gcd(m,n)
```

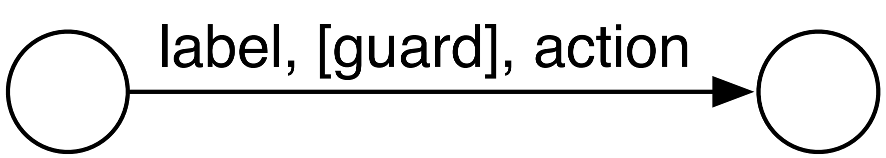


There is a canonical transformation

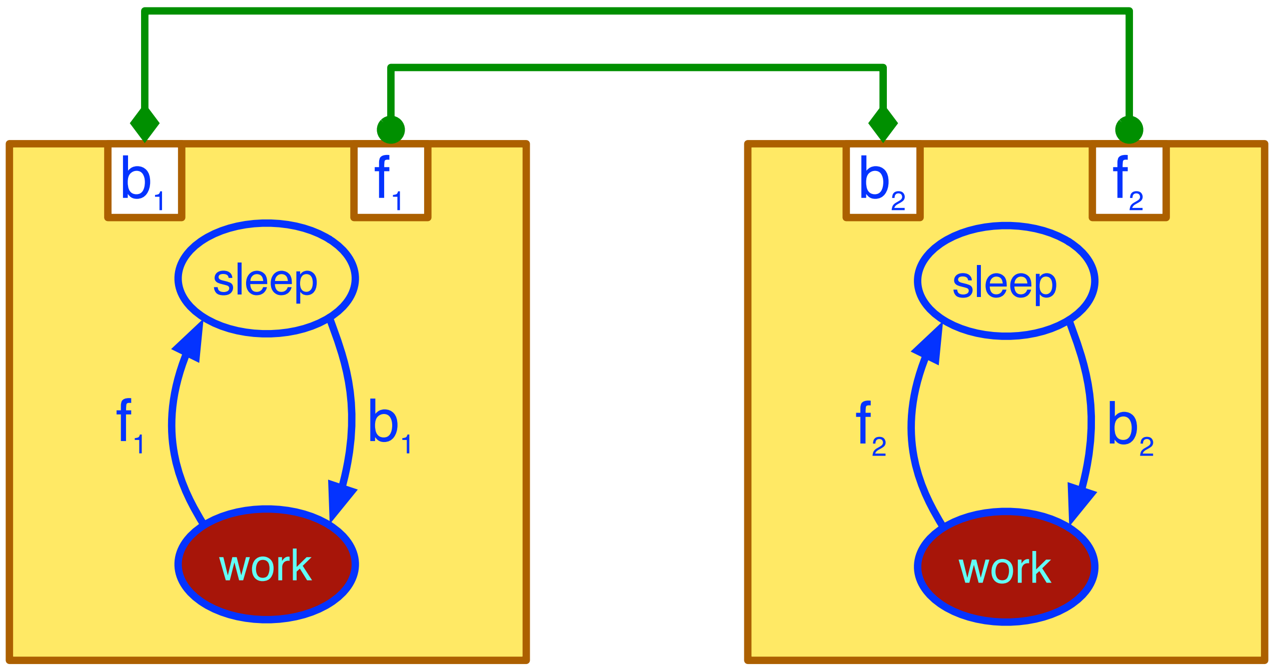
The choice of abstraction level is important

Taking a transition

1. is allowed if the guard evaluates to true
2. executes the action
3. updates current state



BIP by example: Mutual exclusion



Interaction model:

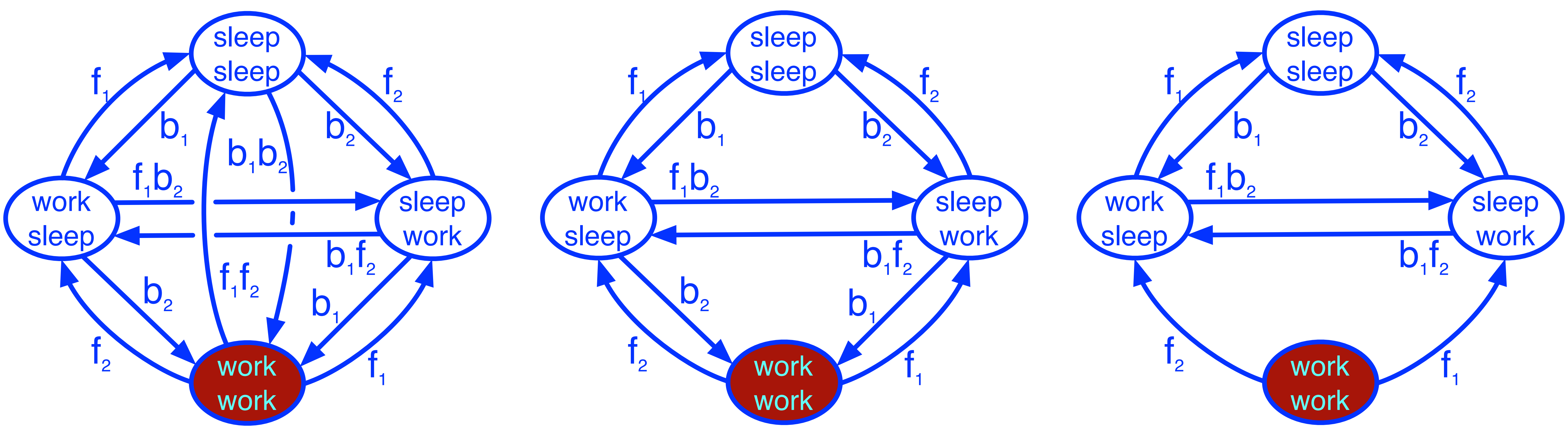
$$\{b_1,\ f_1,\ b_2,\ f_2,\ b_1 f_2,\ b_2 f_1\}$$

Maximal progress:

$$b_1 < b_1 f_2,\ b_2 < b_2 f_1$$

Design view

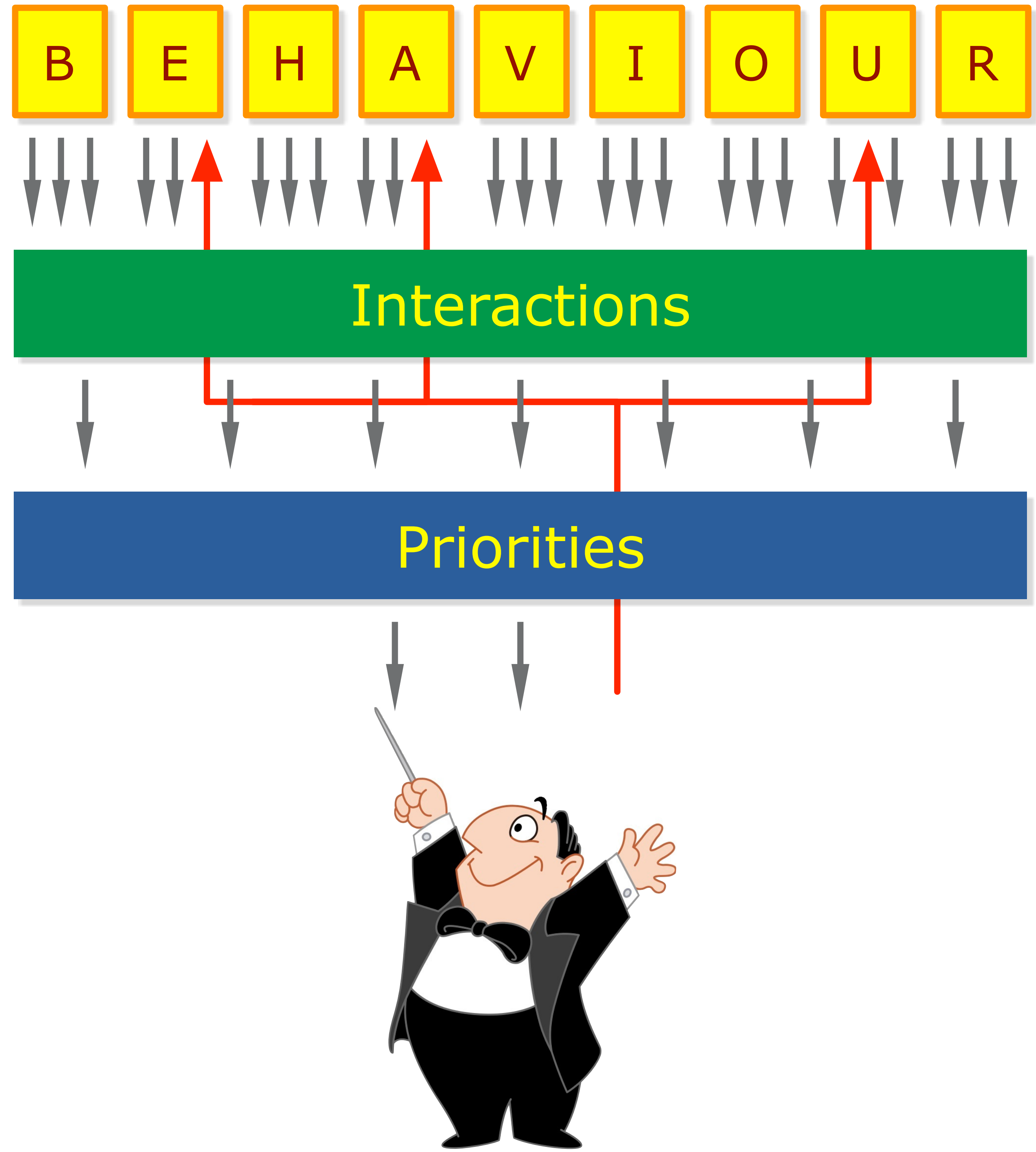
Semantic view

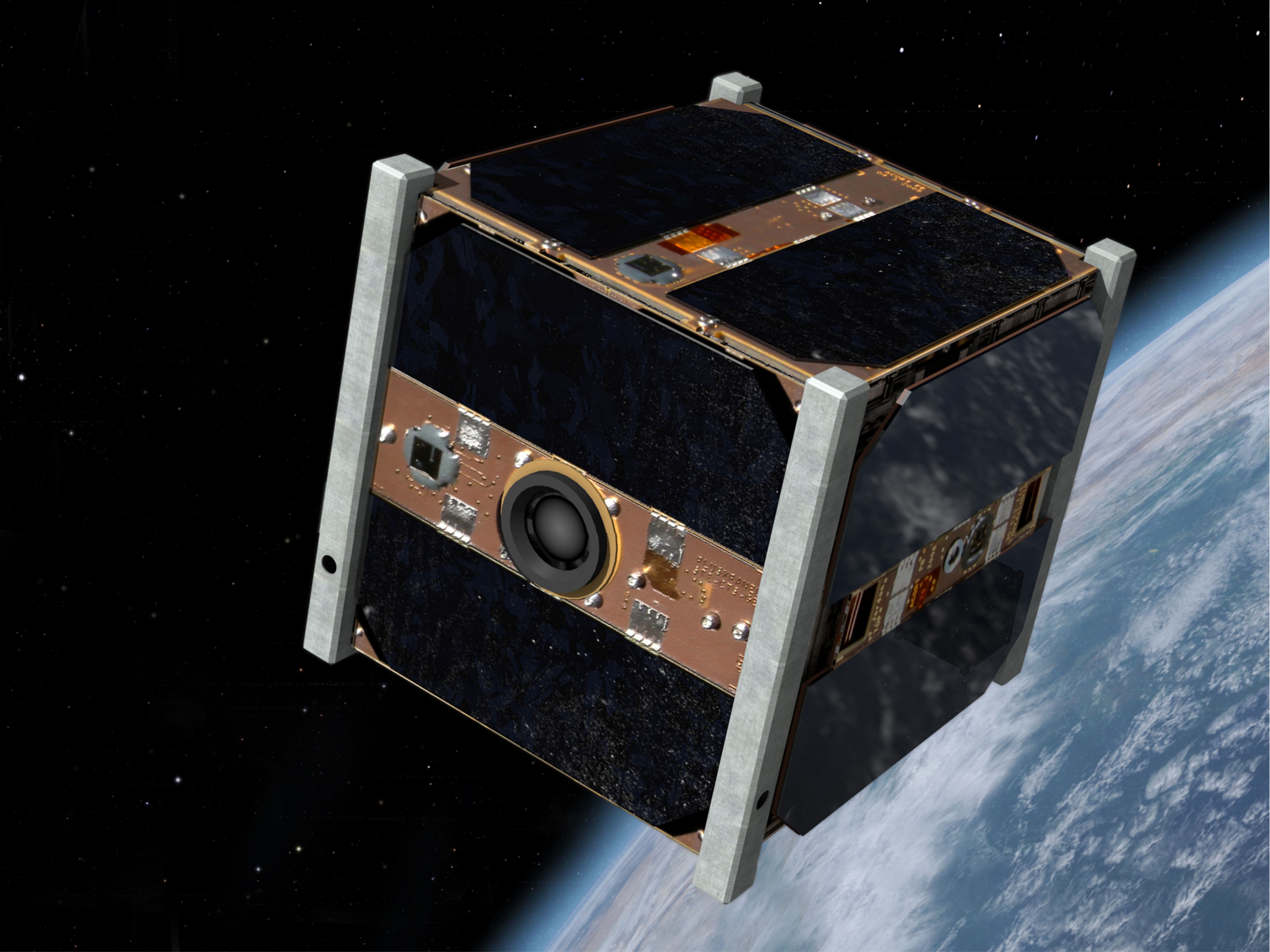


Engine-based execution

1. Components notify the Engine about enabled transitions.

2. The Engine picks an interaction and instructs the components.





Satellite software design

A collaboration with the EPFL Space Engineering Center

- Component-based design in BIP of the control software for a nano-satellite
  - Control and Data Management System (CDMS)
  - Communication with other subsystems through an I^{2}C bus

A collaboration with ThalesAlenia Space (France) and Aristotle University of Thessaloniki (Greece)

- "Catalogue of System and Software Properties"
  - Funded by ESA

Satellite software design

A collaboration with the EPFL Space Engineering Center

- Component-based design in BIP of the control software for a nano-satellite
  - Control and Data Management System (CDMS)
  - Communication with other subsystems through an I^2C bus

A collaboration with ThalesAlenia Space (France) and Aristotle University of Thessaloniki (Greece)

- "Catalogue of System and Software Properties"
  - Funded by ESA

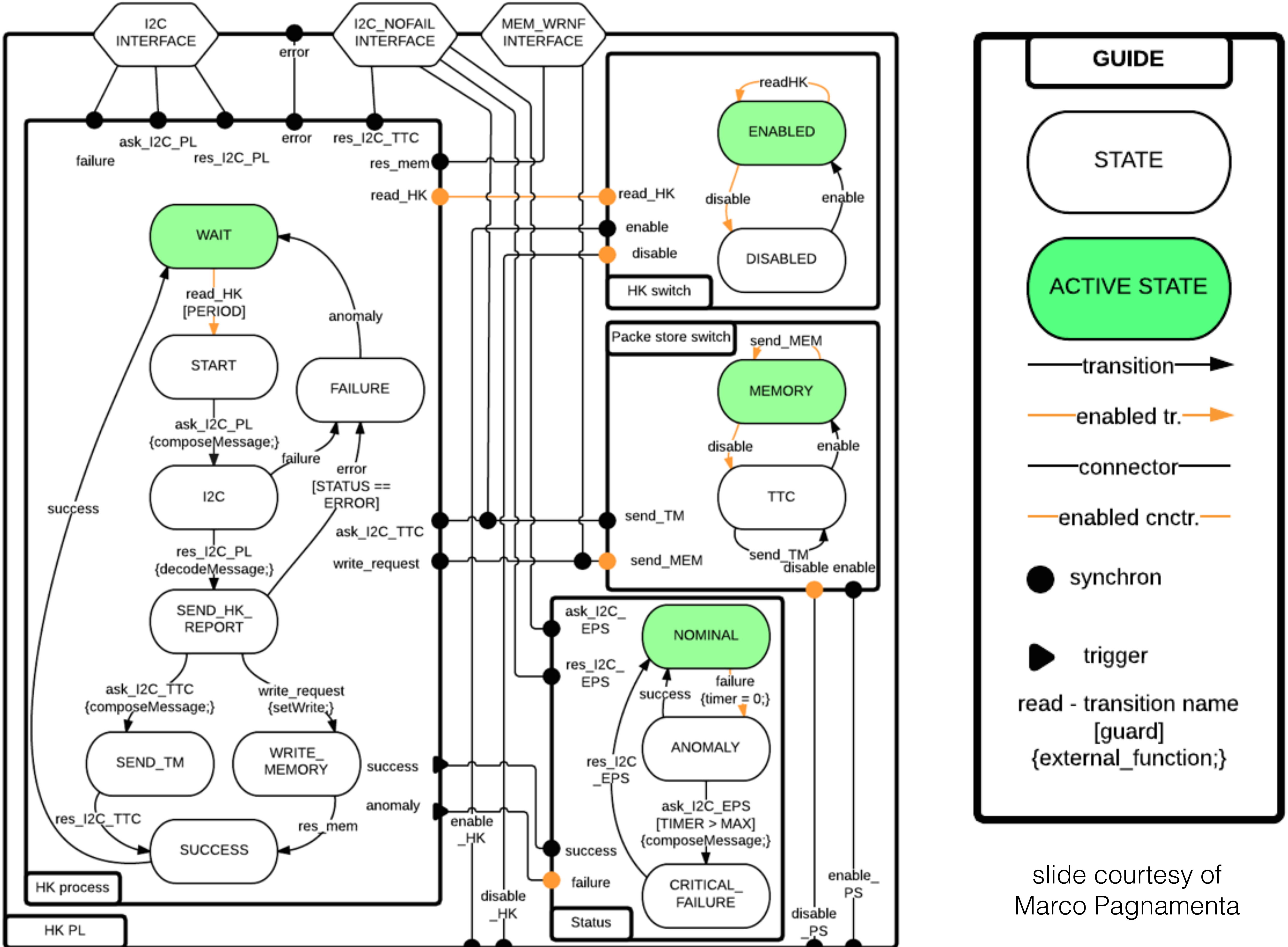
eSpace SPACE ENGINEERING CENTER

ThalesAlenia Space — A Thales / Finmeccanica Company

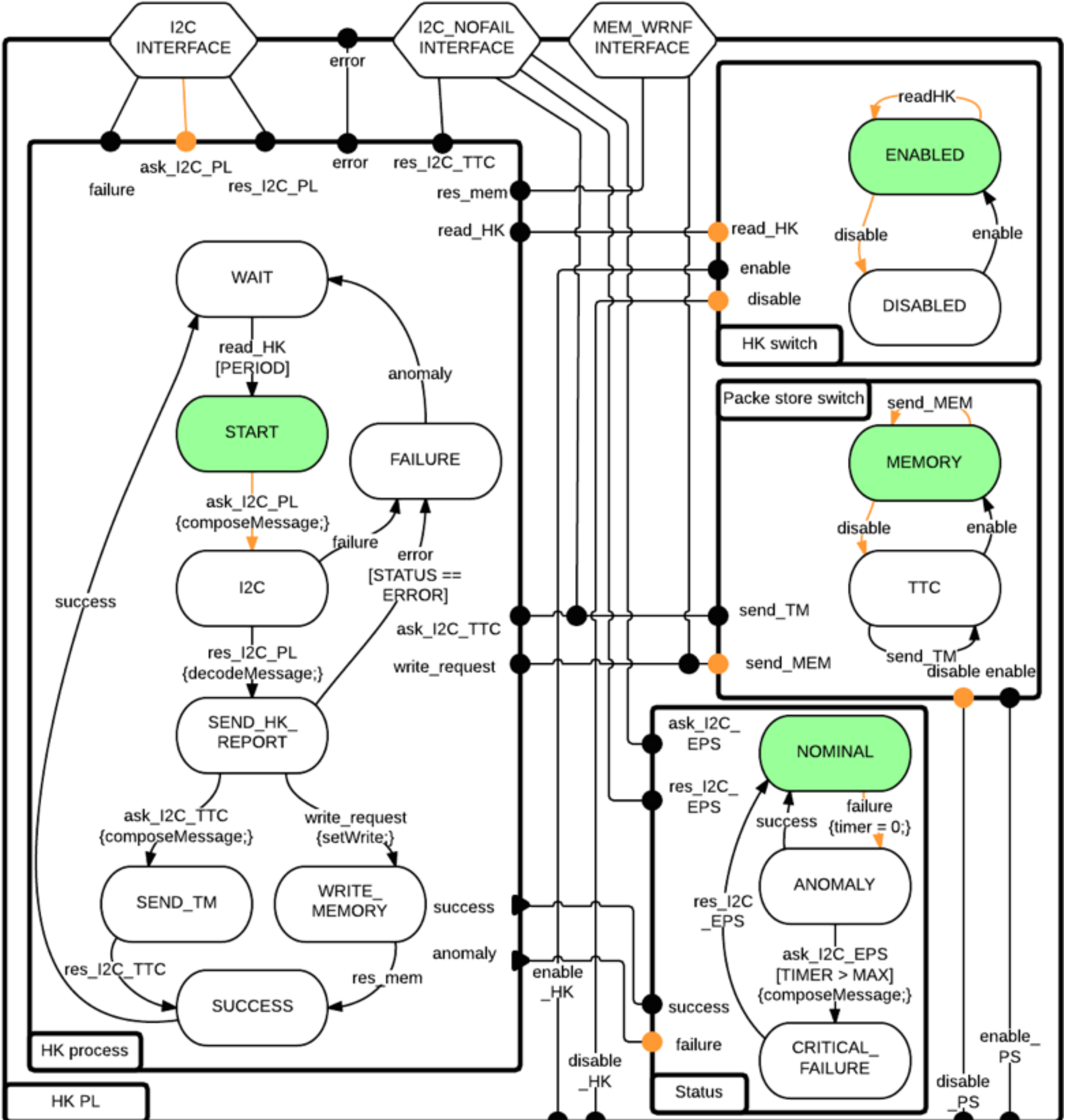
ARISTOTLE UNIVERSITY OF THESSALONIKI

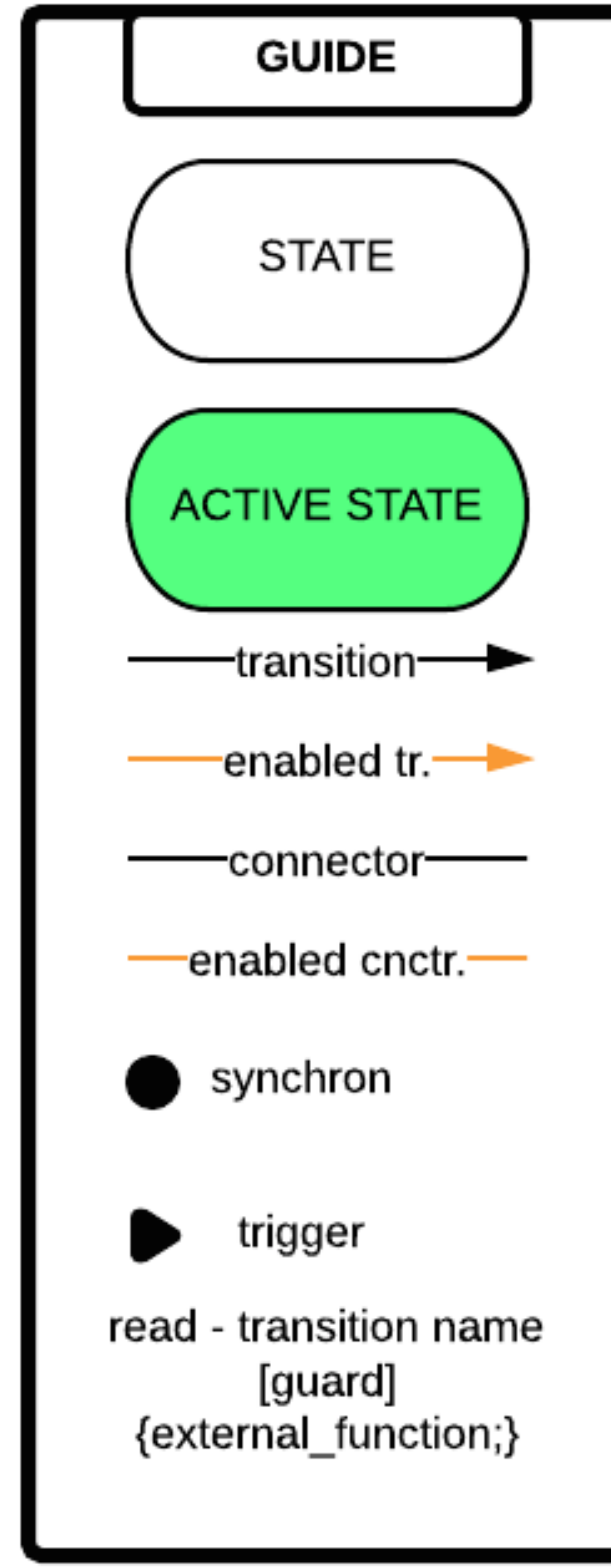
Example 1

Nominal housekeeping routine

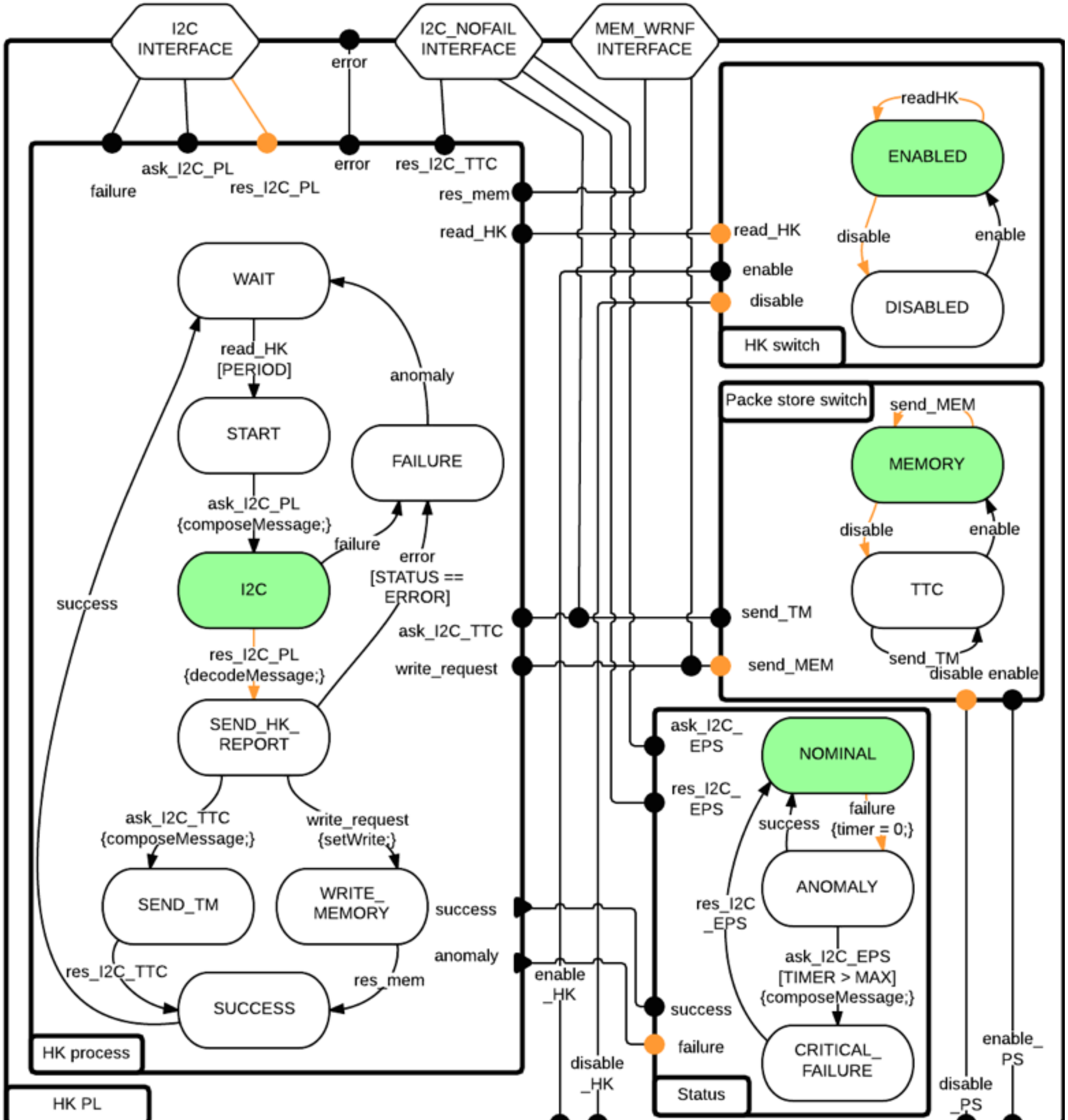


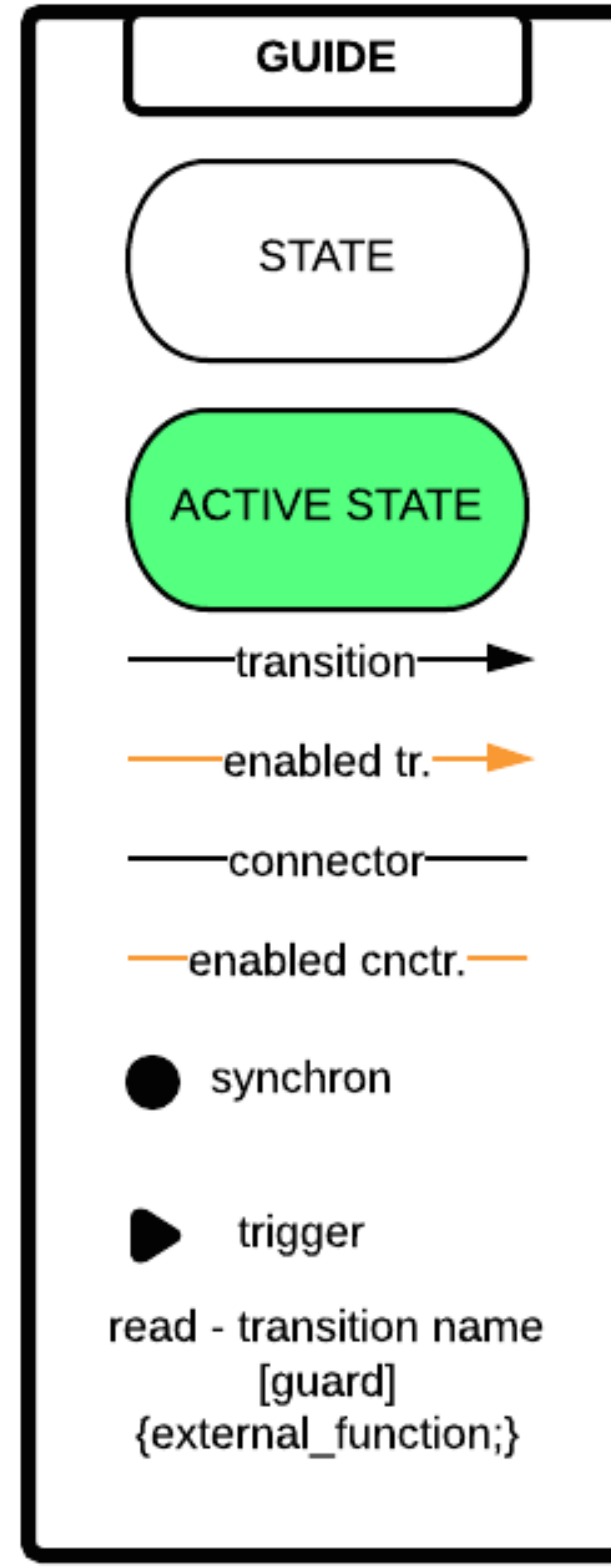
slide courtesy of Marco Pagnamenta



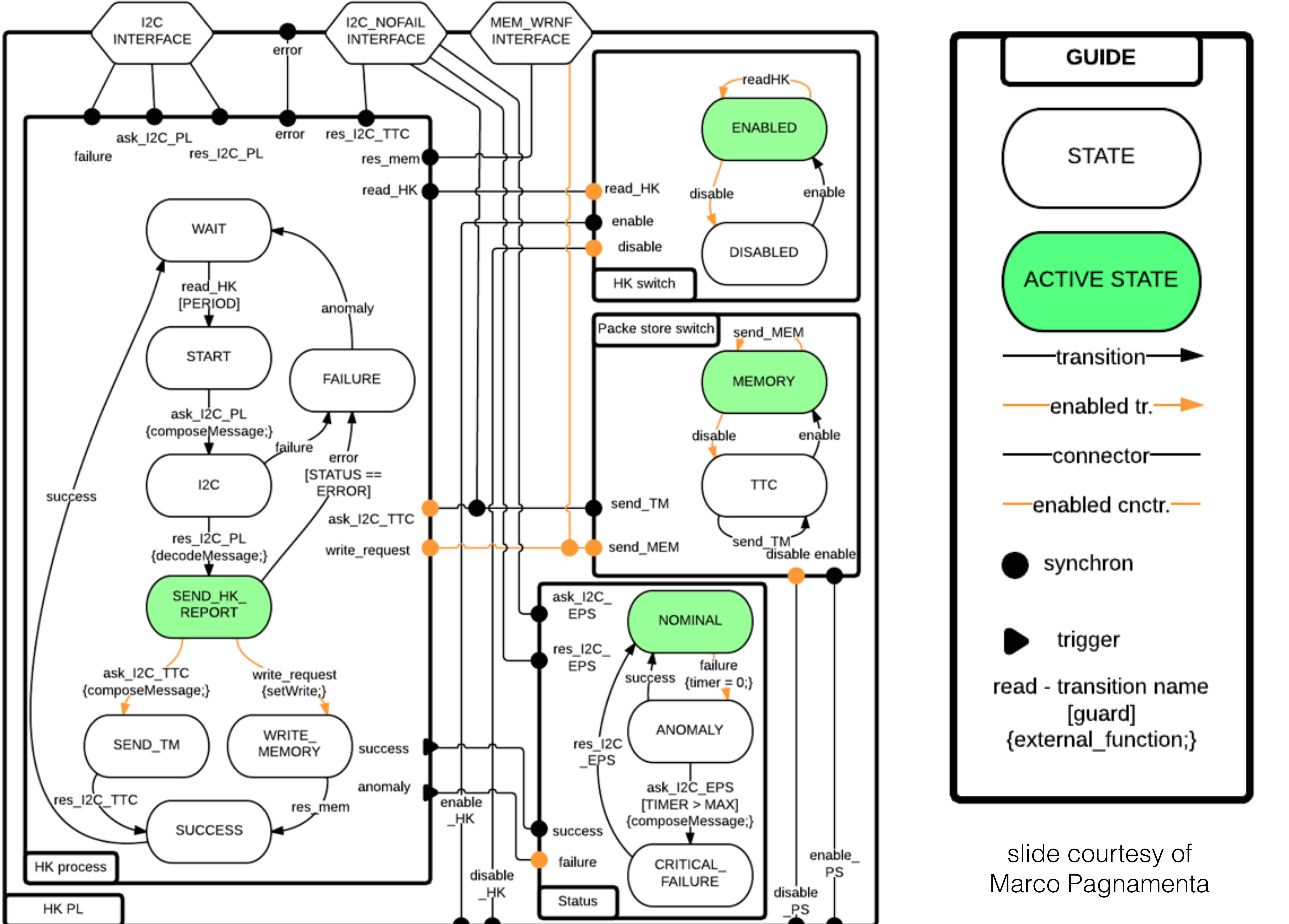


slide courtesy of Marco Pagnamenta

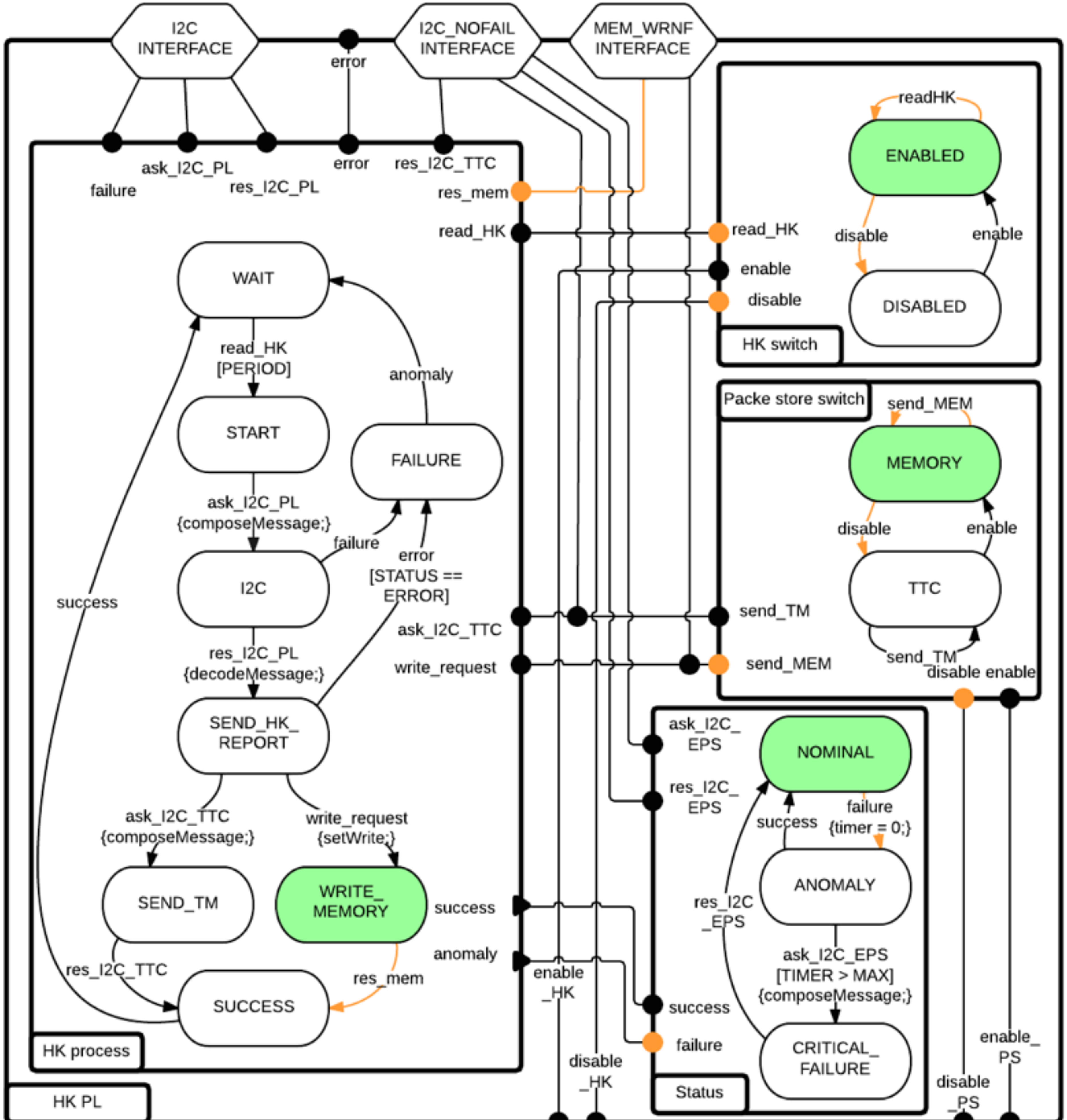


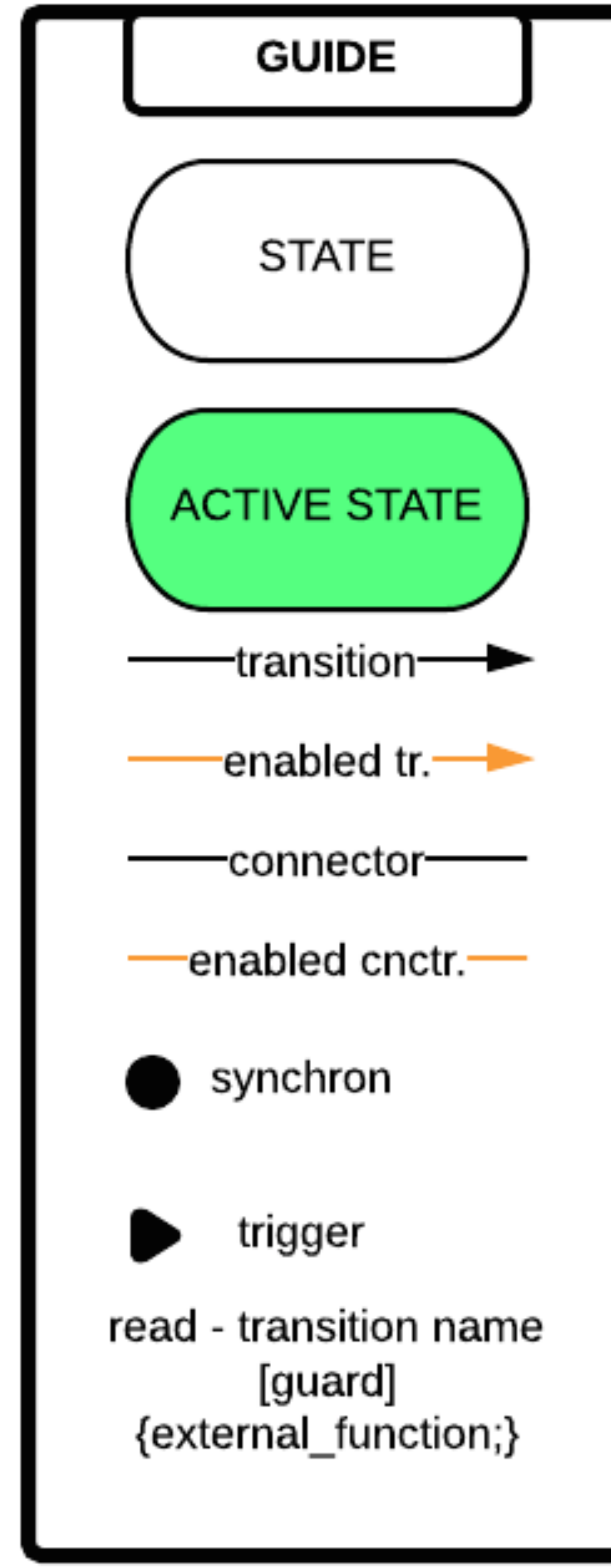


slide courtesy of
Marco Pagnamenta

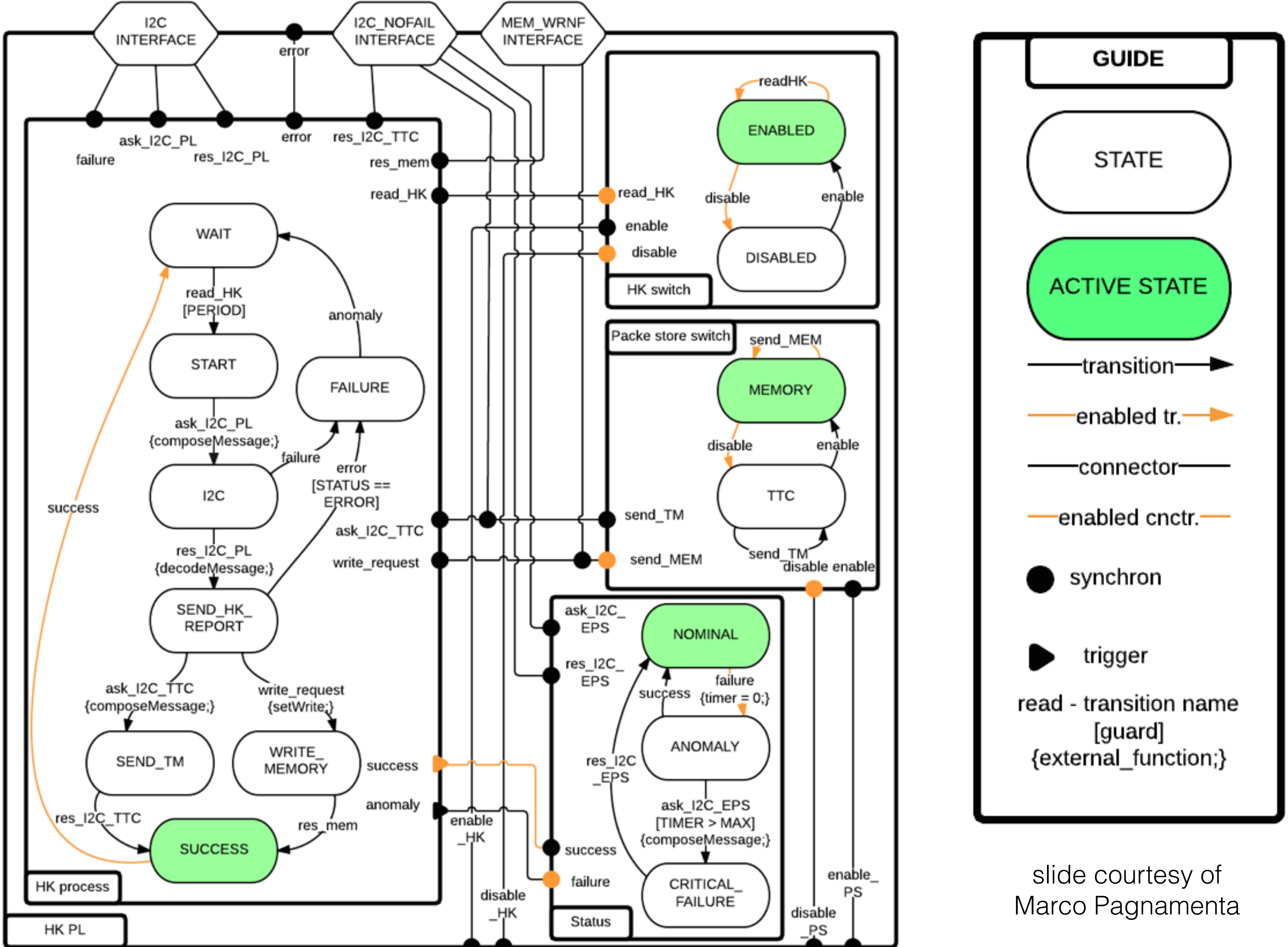


slide courtesy of Marco Pagnamenta





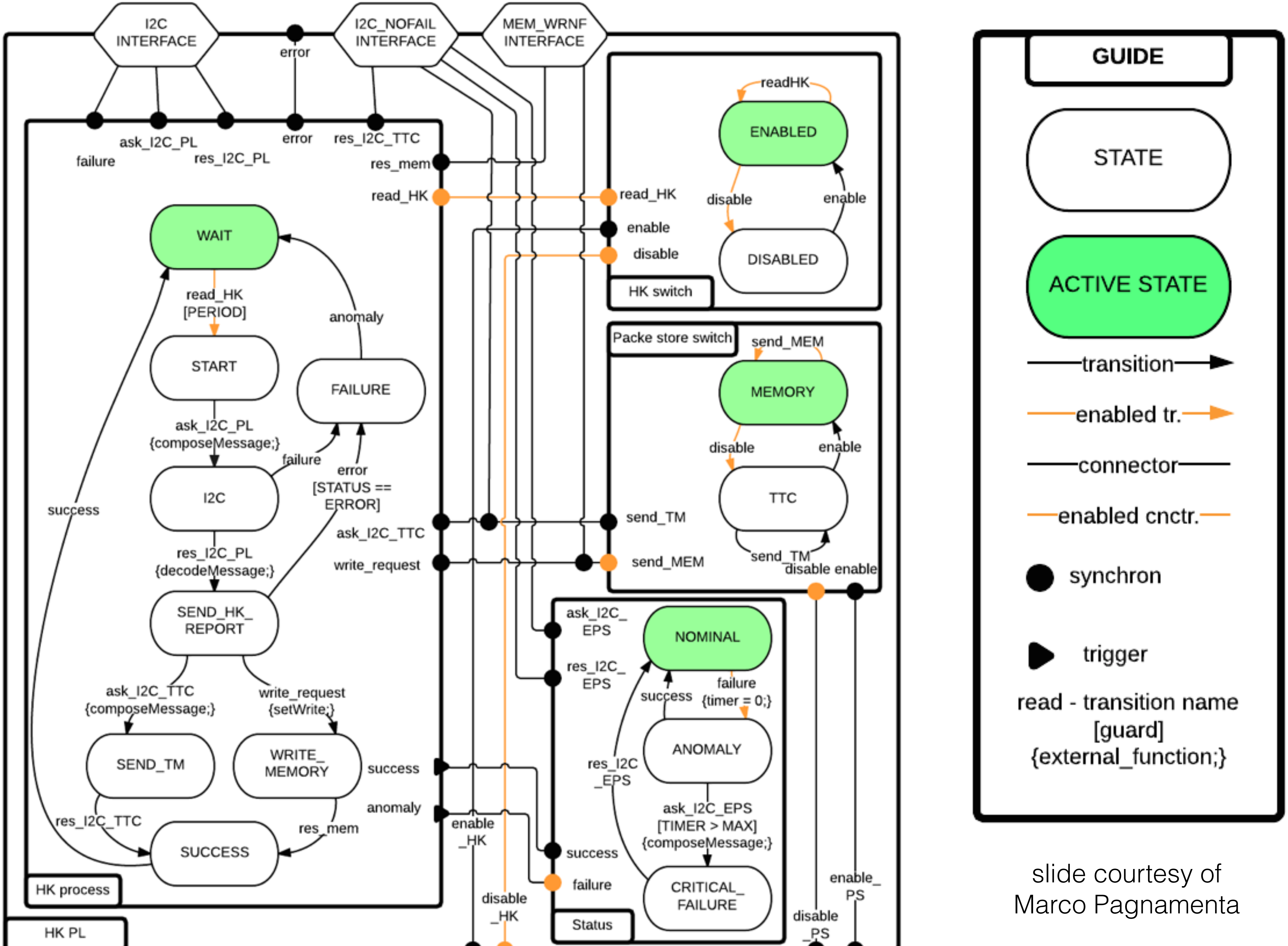
slide courtesy of
Marco Pagnamenta



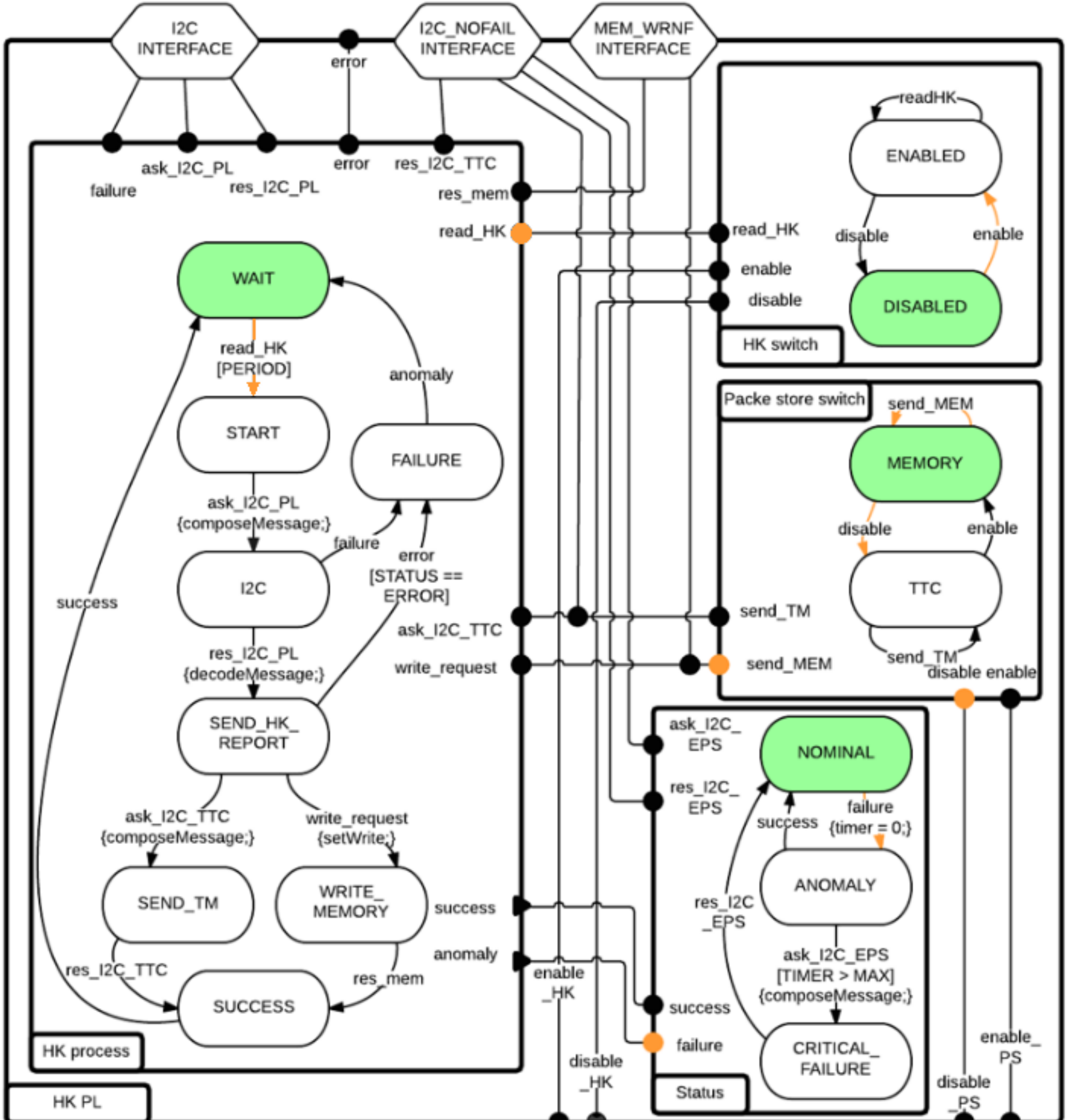
slide courtesy of Marco Pagnamenta

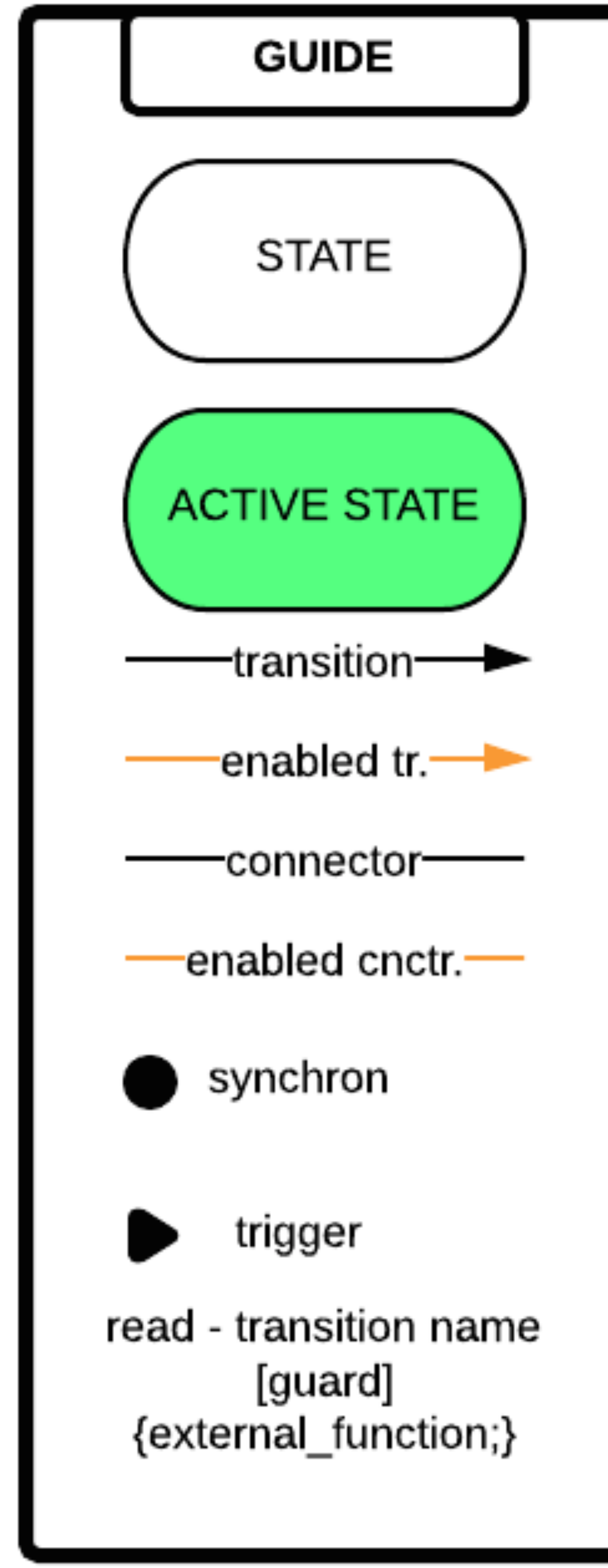
Example 2

Stopping housekeeping



slide courtesy of Marco Pagnamenta

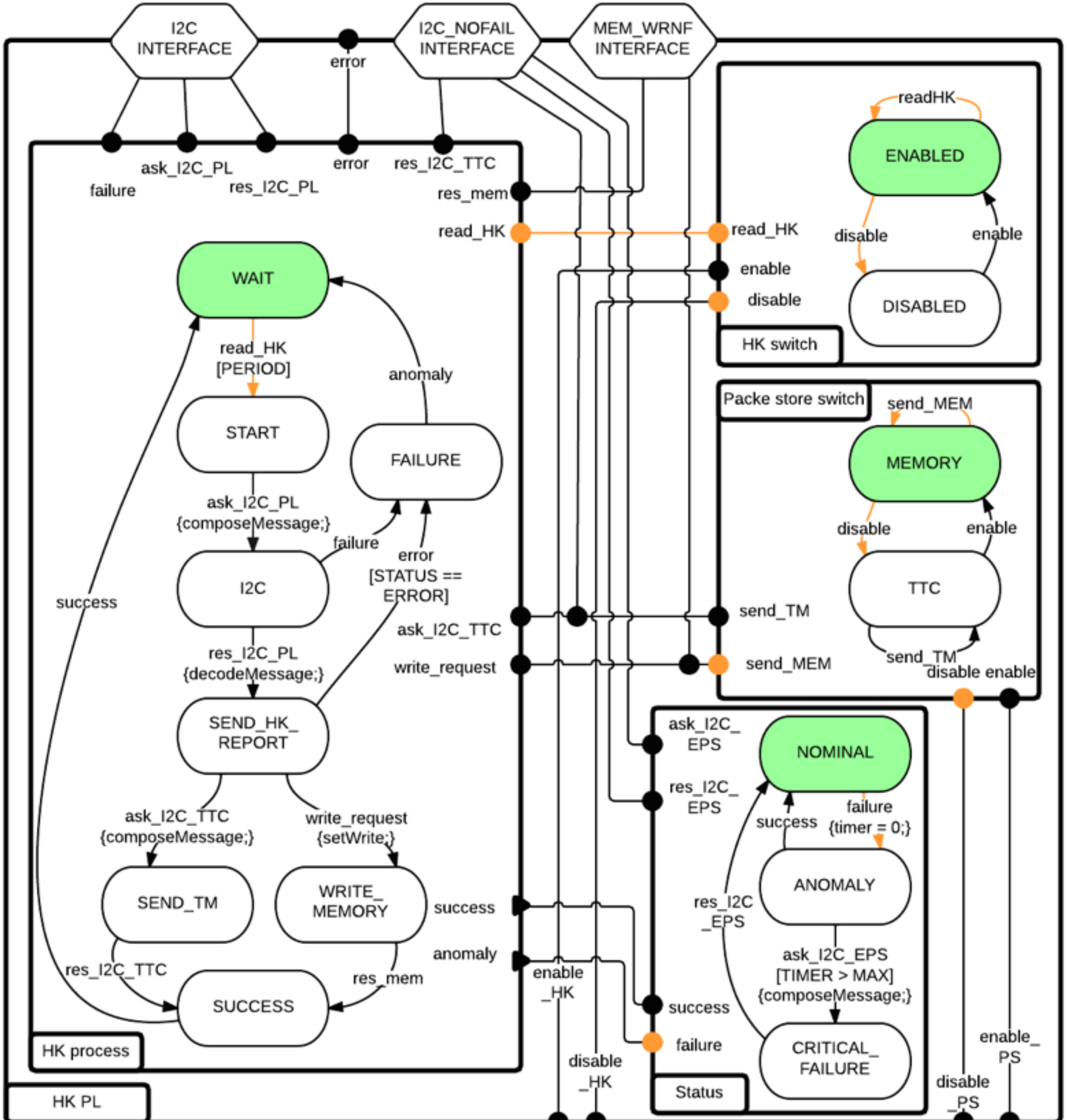


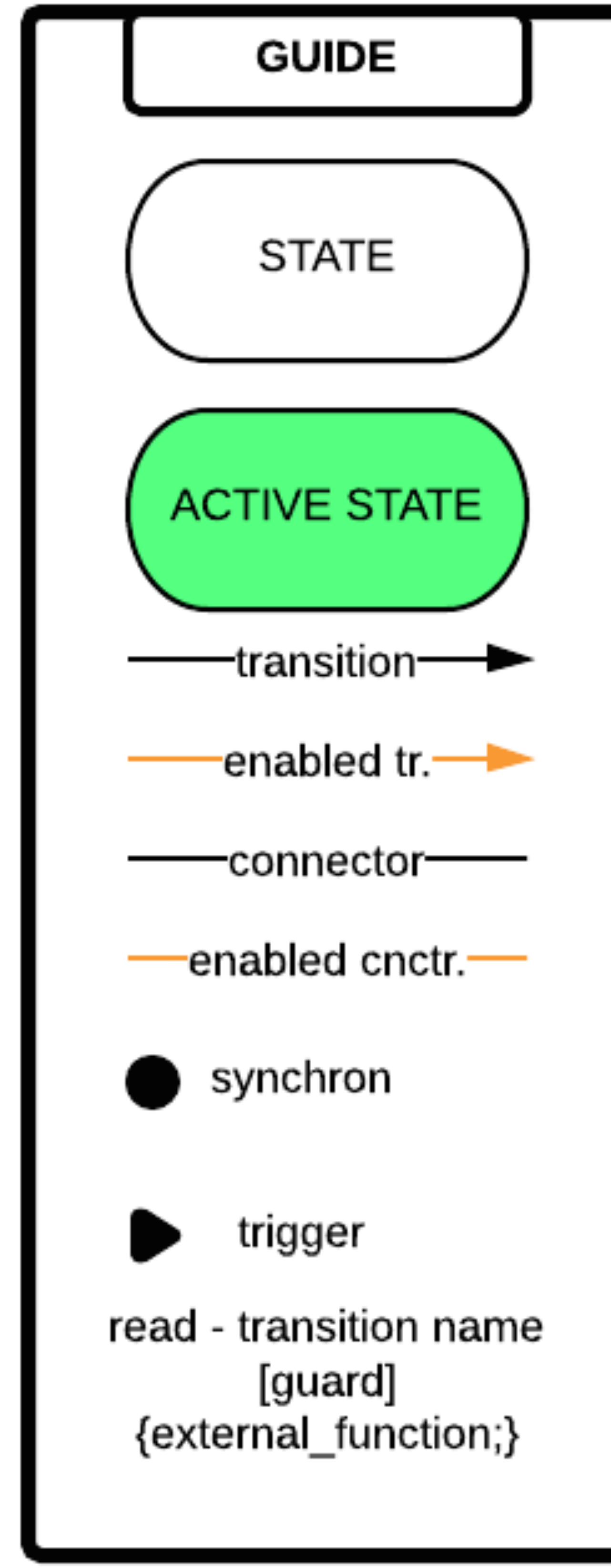


slide courtesy of Marco Pagnamenta

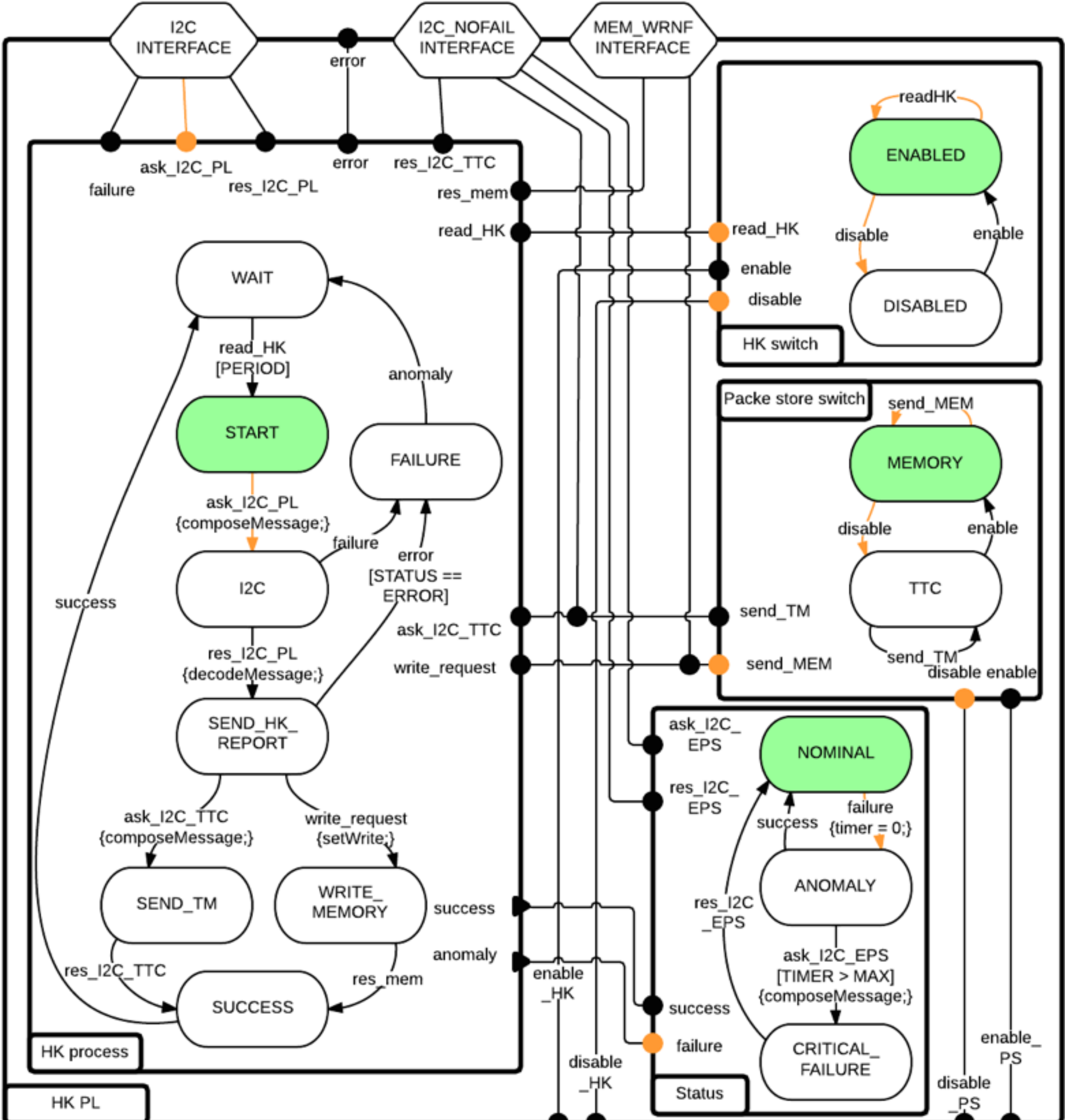
Example 3

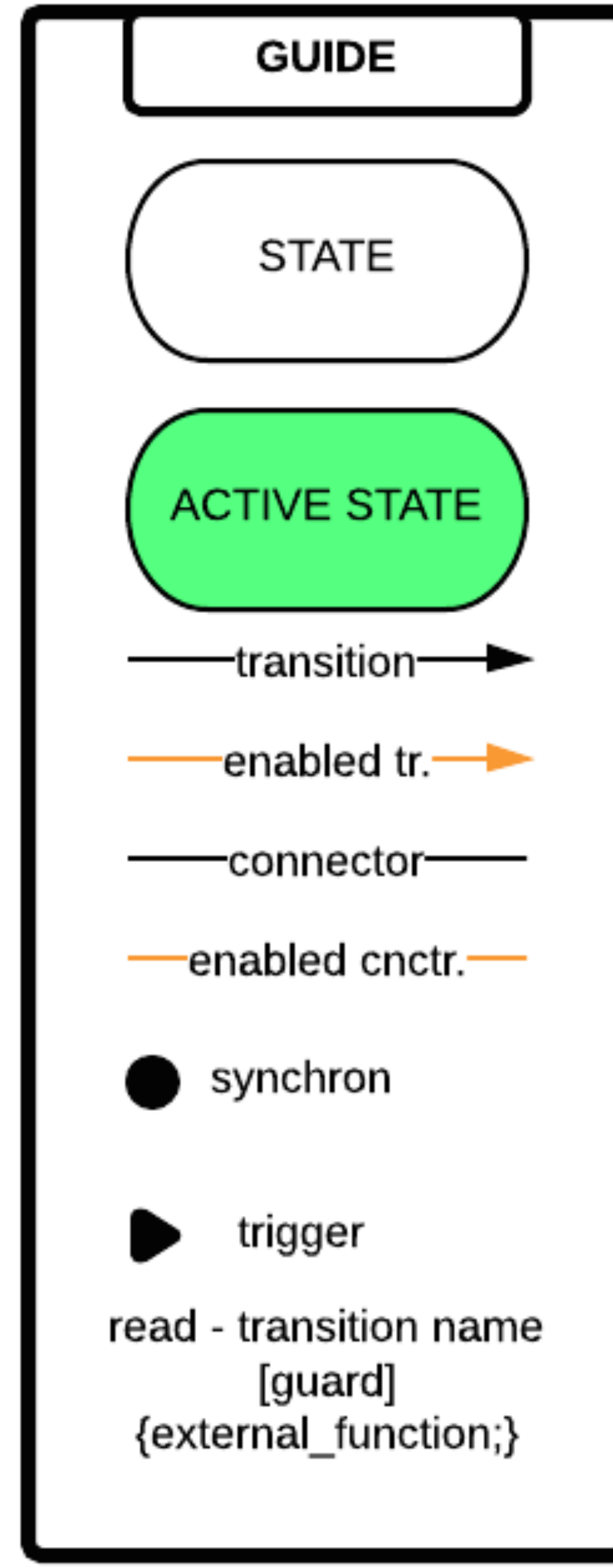
Switching destination of housekeeping data



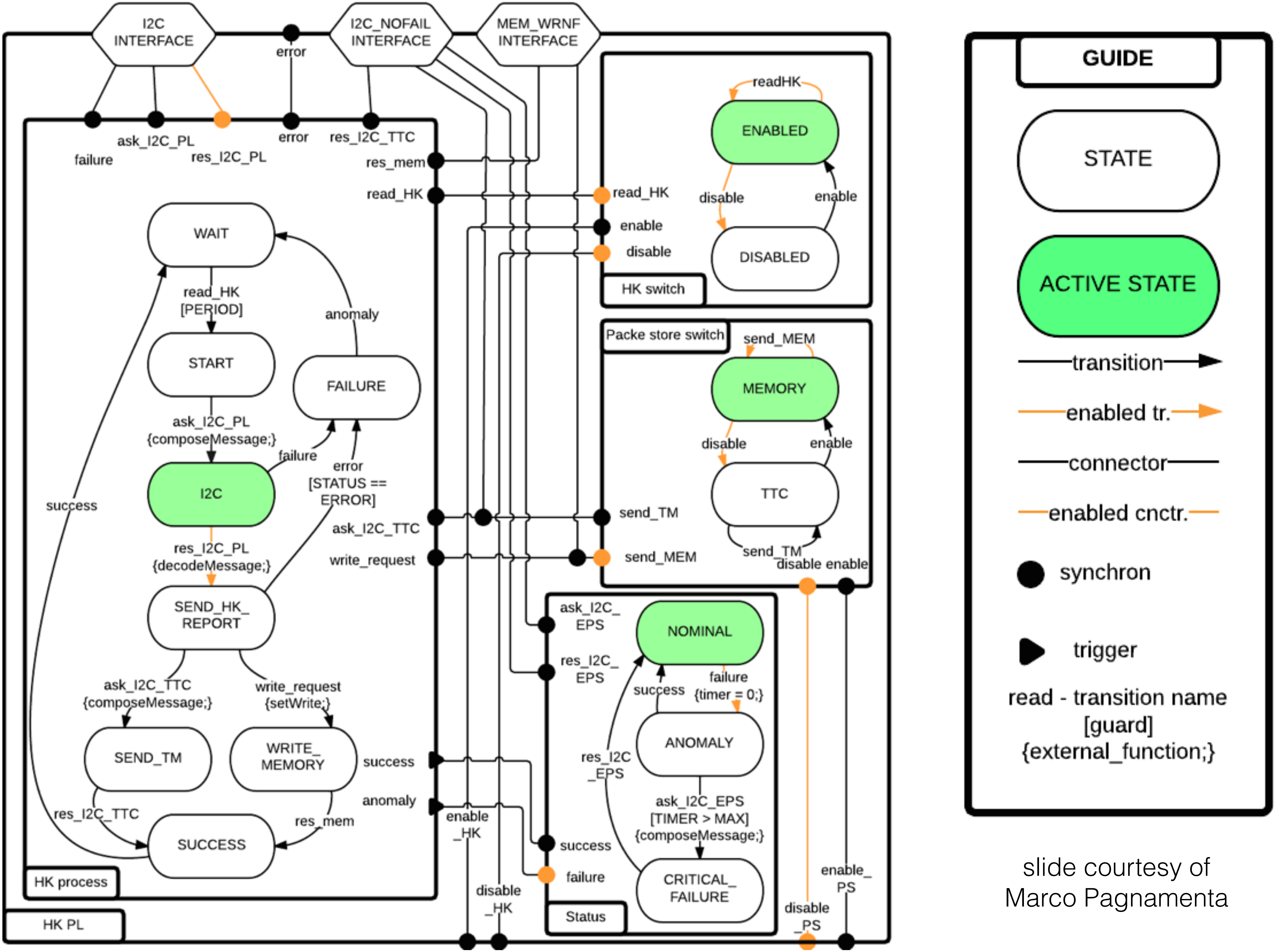


slide courtesy of
Marco Pagnamenta

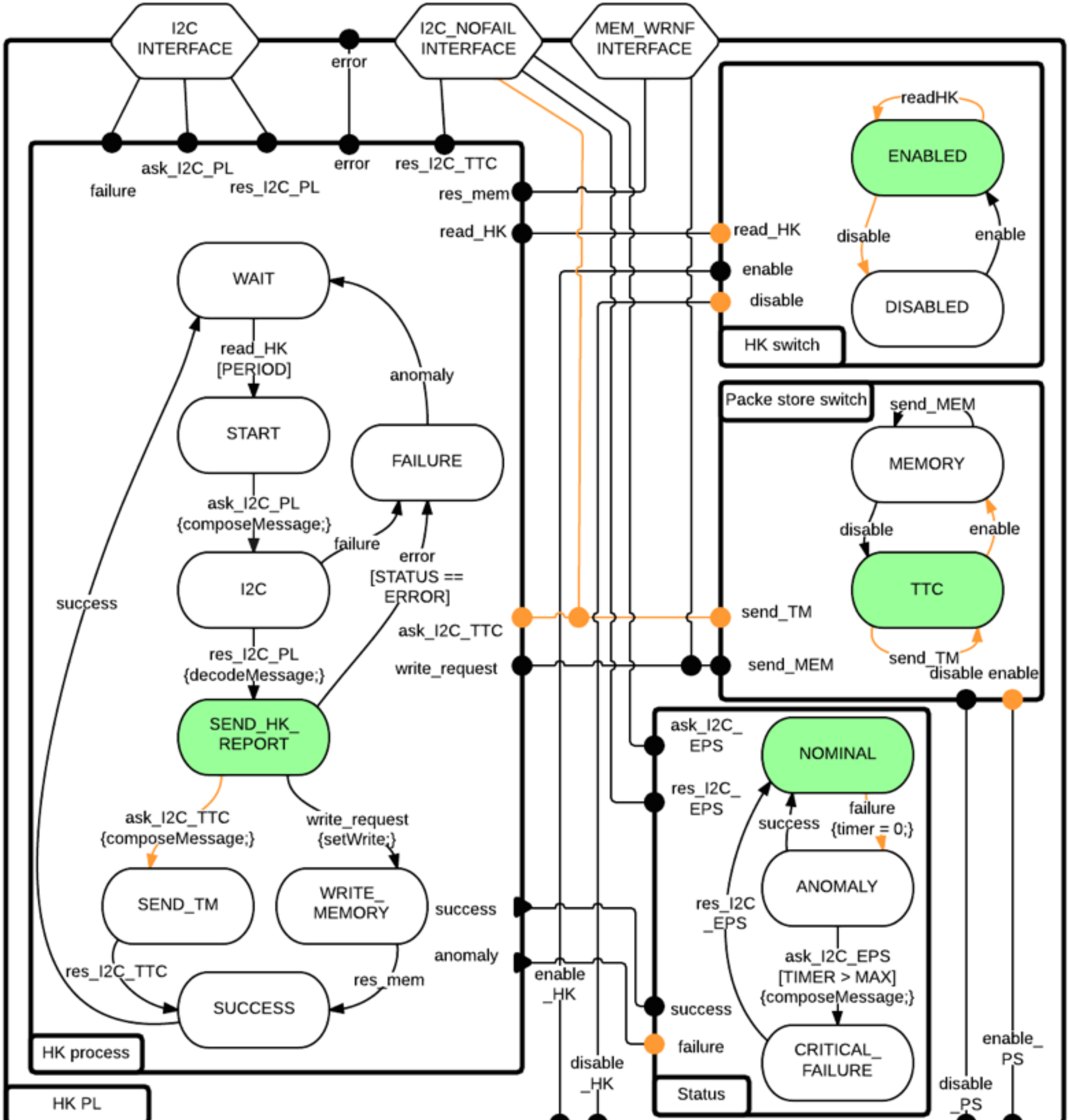


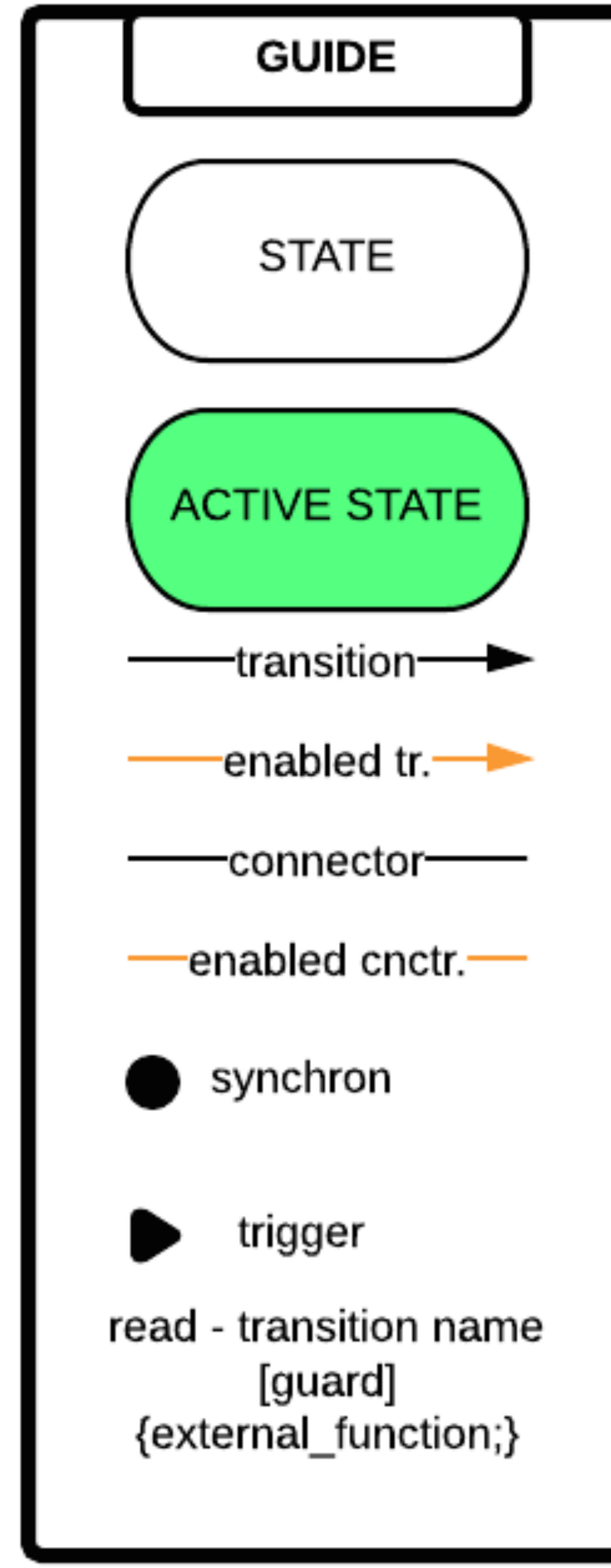


slide courtesy of
Marco Pagnamenta

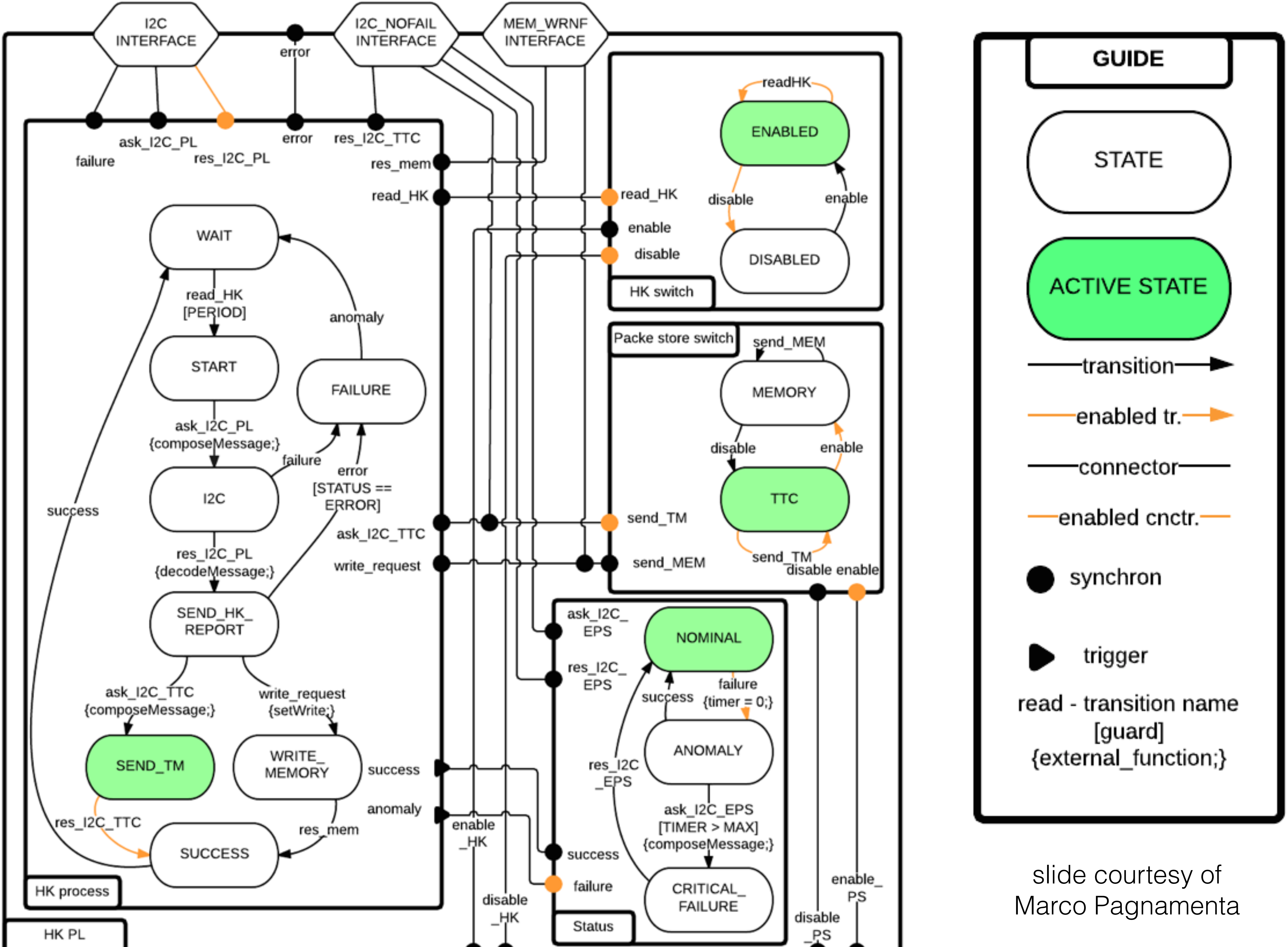


slide courtesy of Marco Pagnamenta

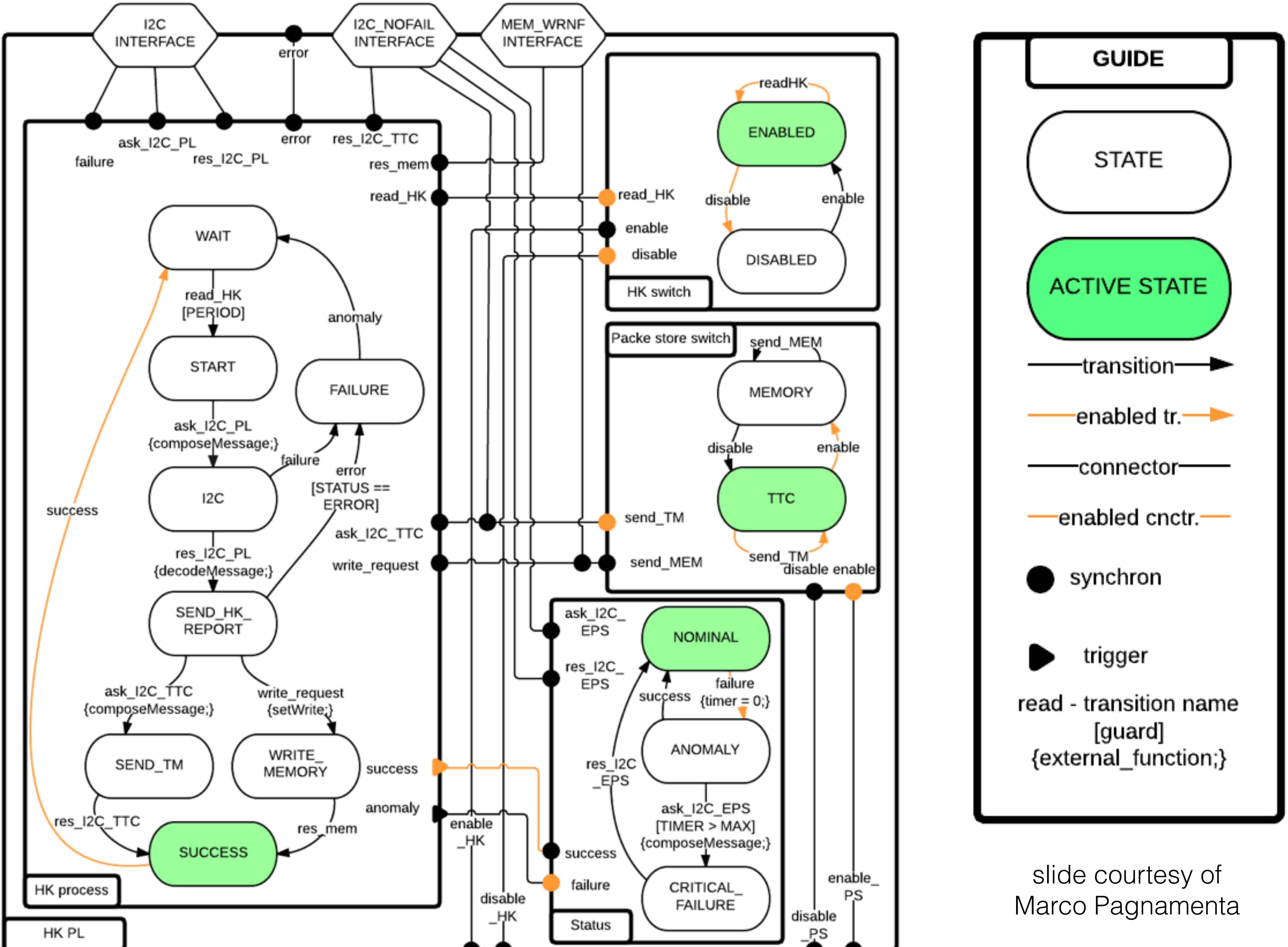




slide courtesy of
Marco Pagnamenta



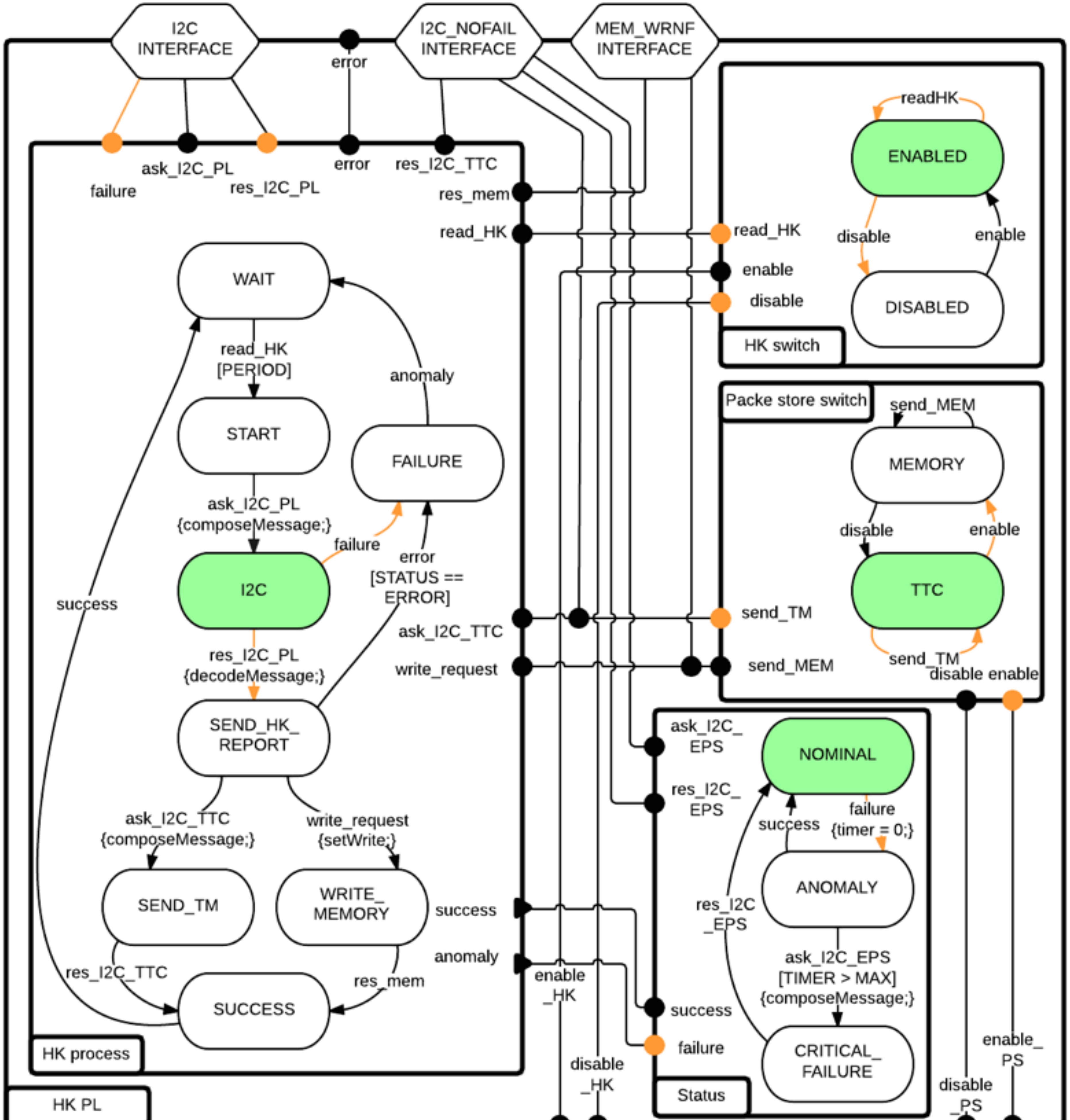
slide courtesy of Marco Pagnamenta

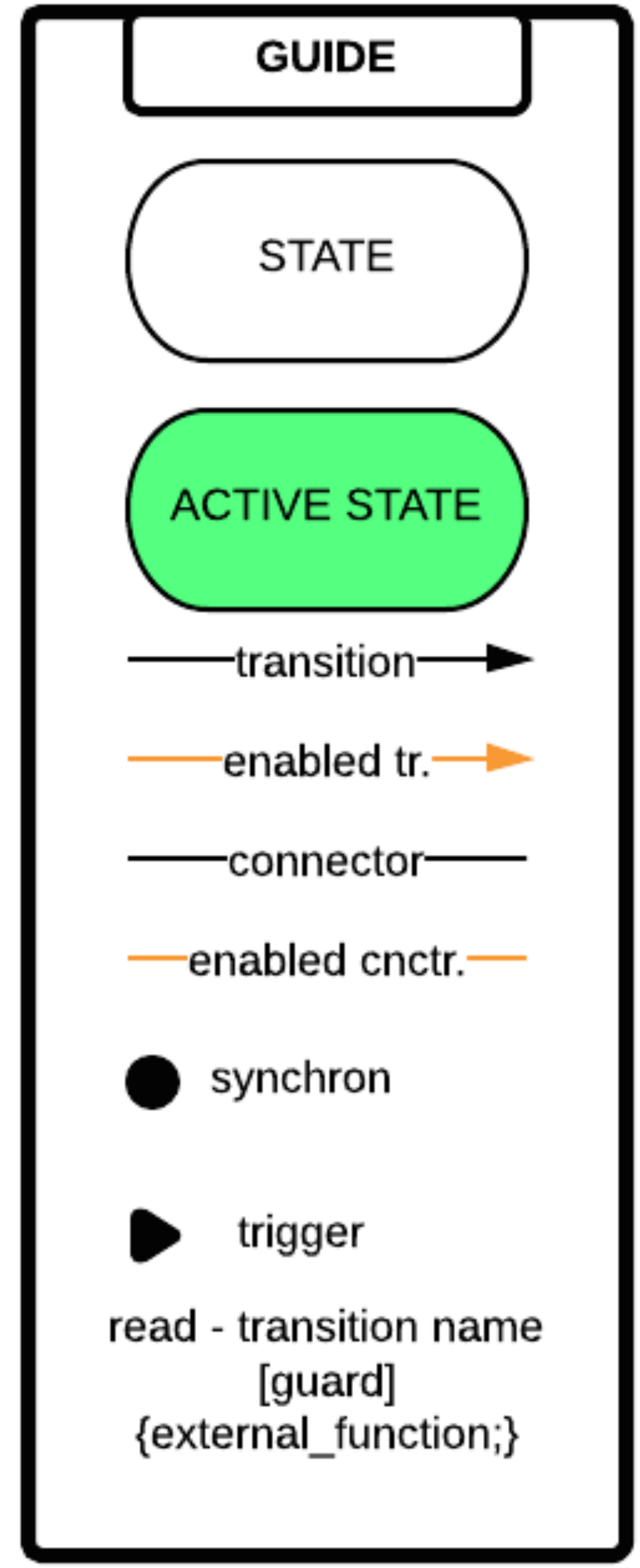


slide courtesy of Marco Pagnamenta

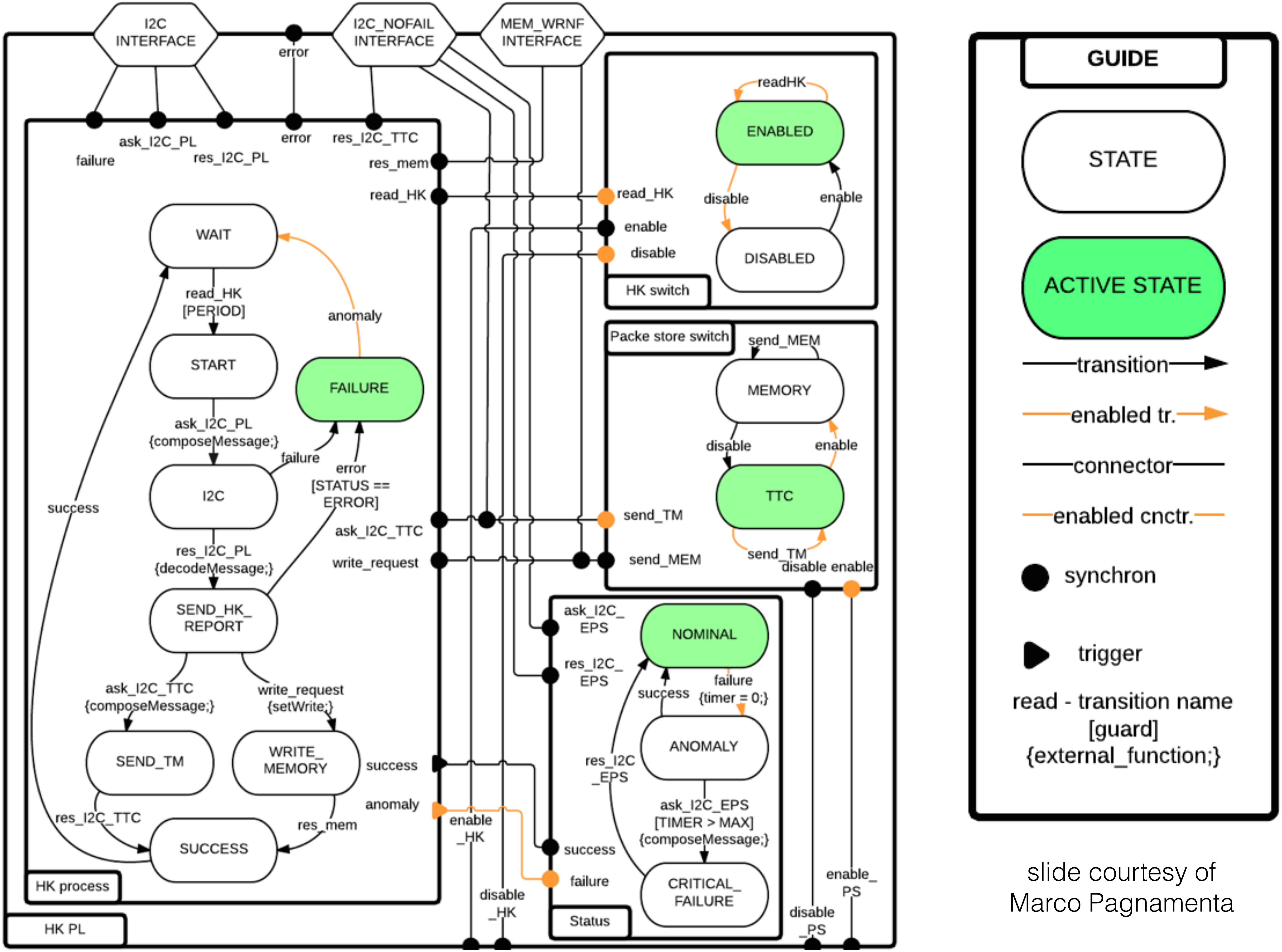
Example 4

I^2C bus failure management

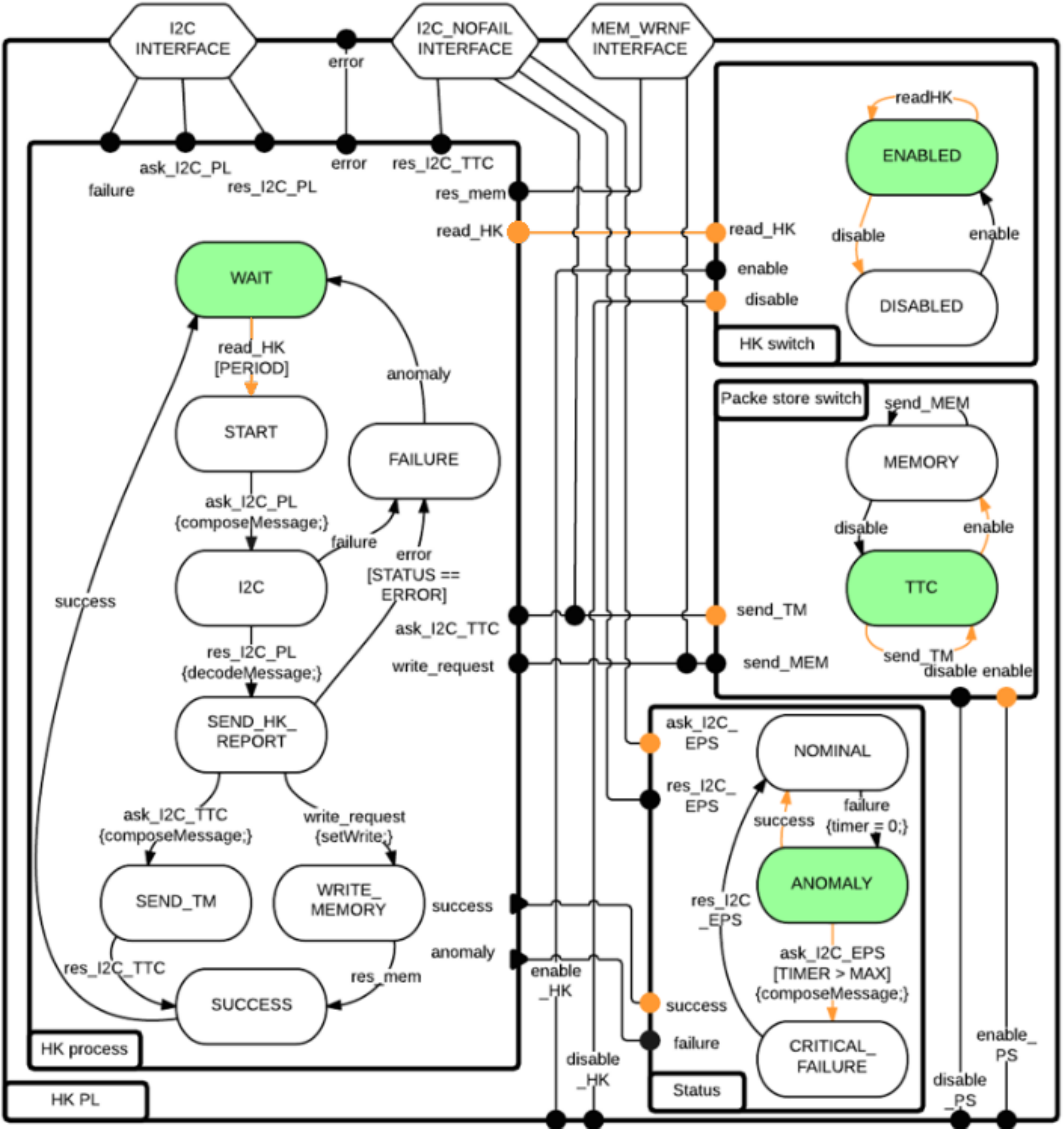


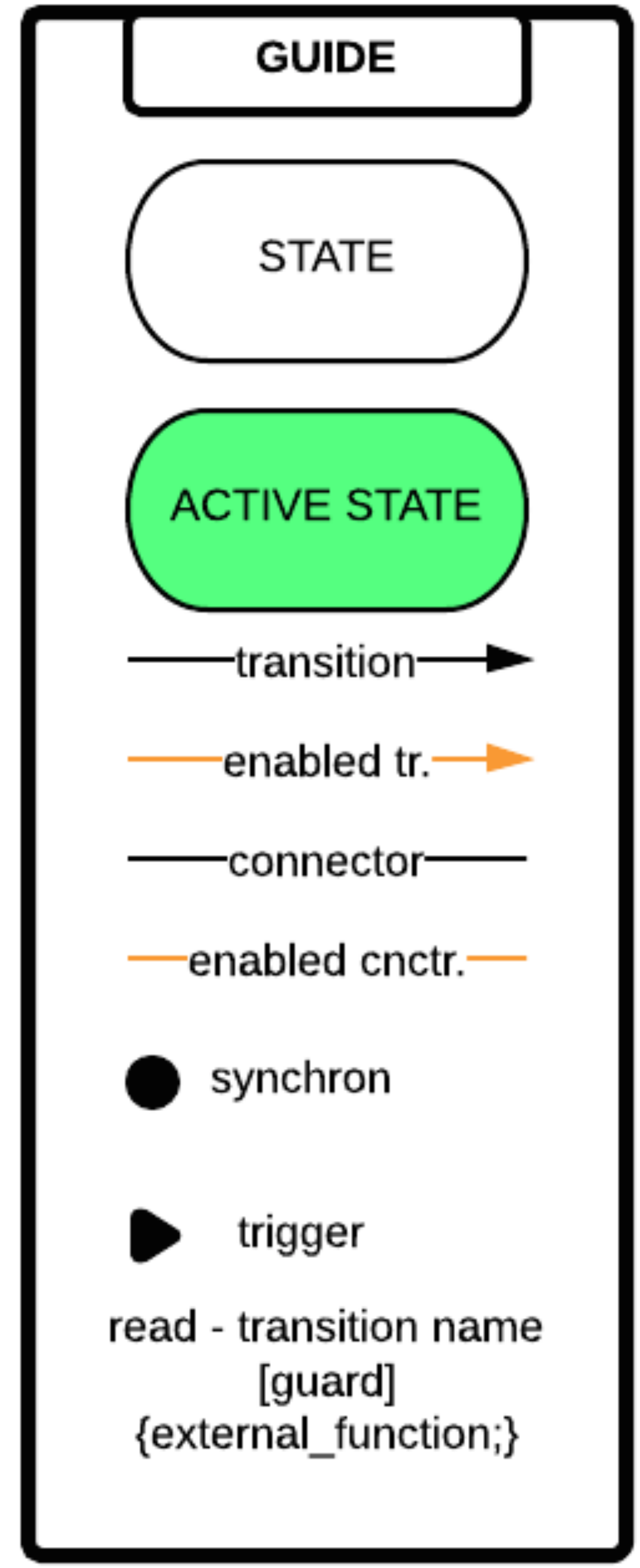


slide courtesy of
Marco Pagnamenta



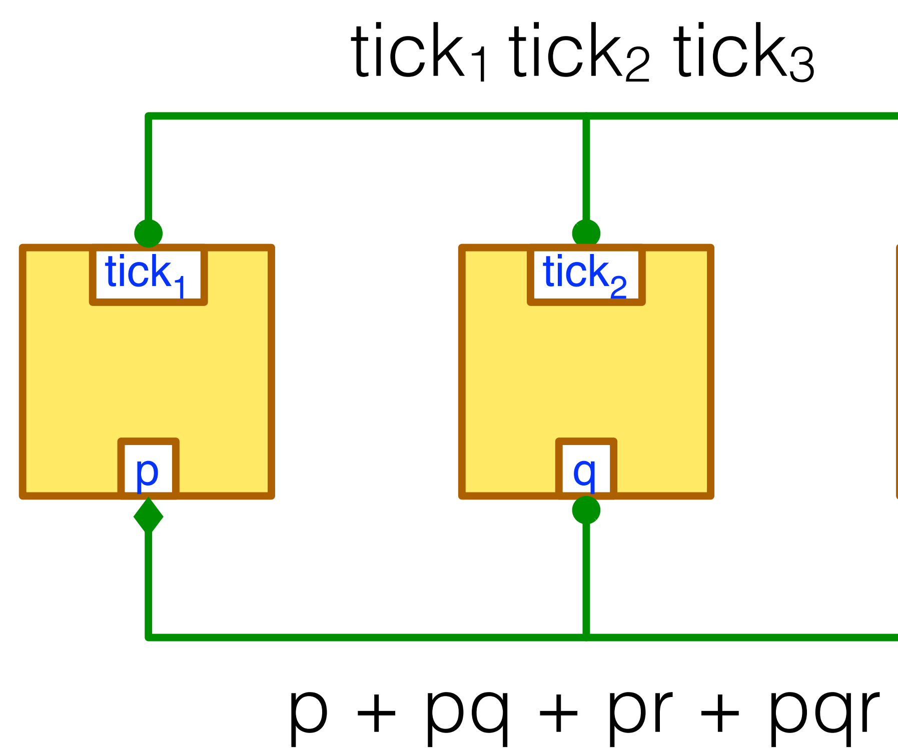
slide courtesy of Marco Pagnamenta





slide courtesy of
Marco Pagnamenta

Connectors



Connectors are tree-like structures

ports as leaves and nodes of two types

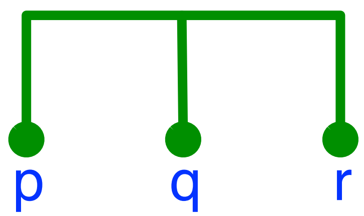
Triggers (diamonds) — nodes that can “initiate” an interaction

Synchrons (bullets) — nodes that can only “join” an interaction initiated by others

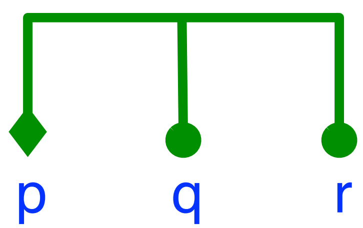
In practice, **maximal progress** is implicitly assumed

Connector examples

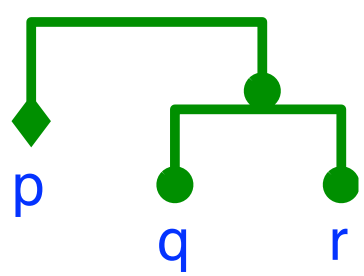
The Algebra of Connectors



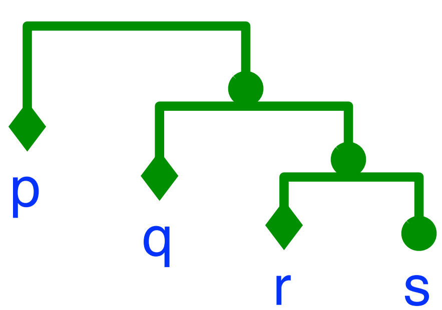
Strong synchronisation: pqr
p q r



Broadcast: p + pq + pr + pqr
p’ q r



Atomic broadcast: p + pqr
p’ [q r]



Causal chain: p + pq + pqr + pqrs
p’ [q’ [r’ s]]

The Algebra of Connectors—Structuring Interaction in BIP
S. Bliudze, J. Sifakis [EMSOFT’07]

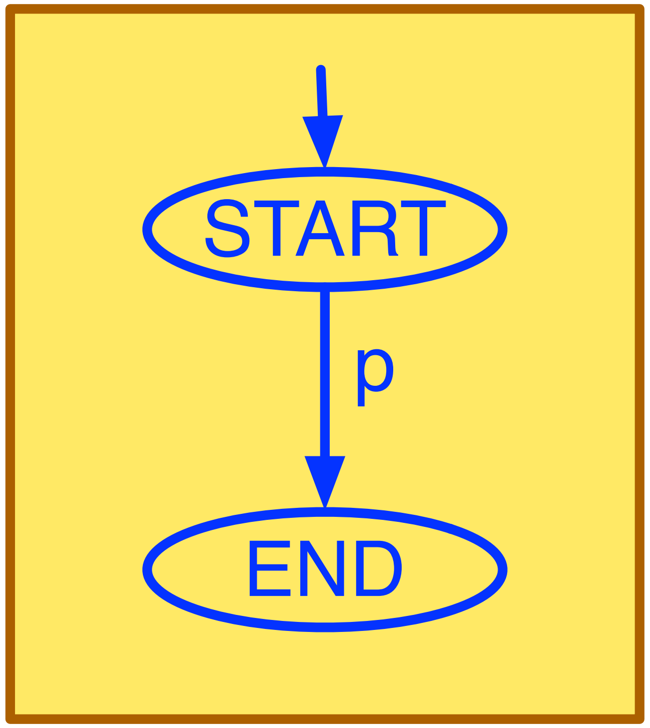
Hands-on BIP

Safe control layer of a Rescue robot

https://www.bliudze.me/simon/auth-bip

Hello World

```
package HelloPackage
  port type HelloPort_t()

  atom type HelloAtom()
    port HelloPort_t p()

    place START,END

    initial to START
    on p from START to END
  end

  compound type HelloCompound()
    component HelloAtom c1()
  end
end
```



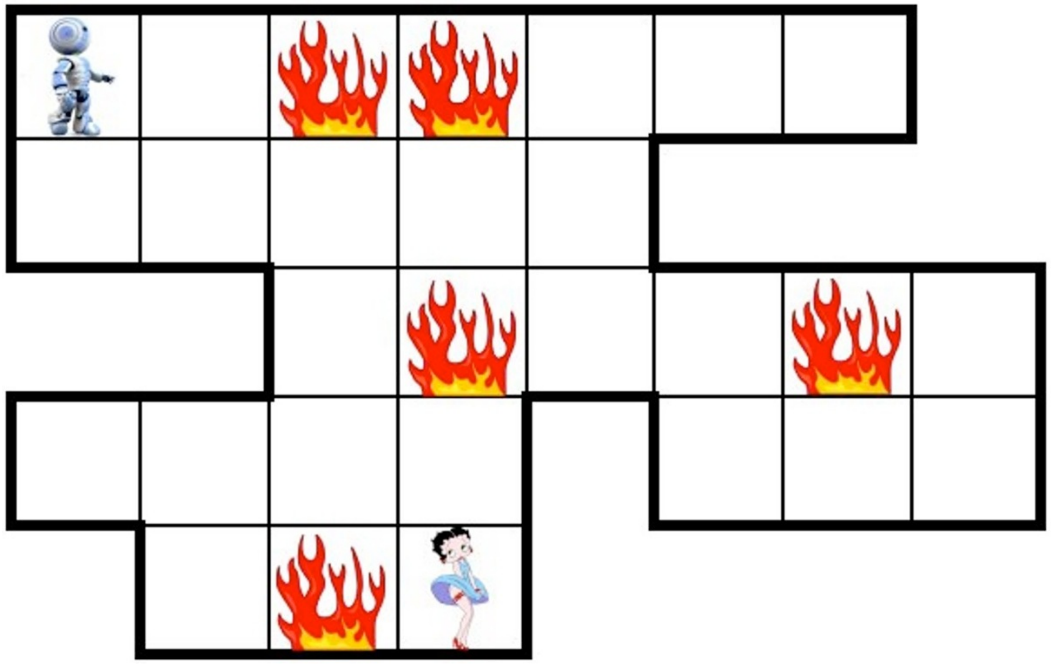
Hello World

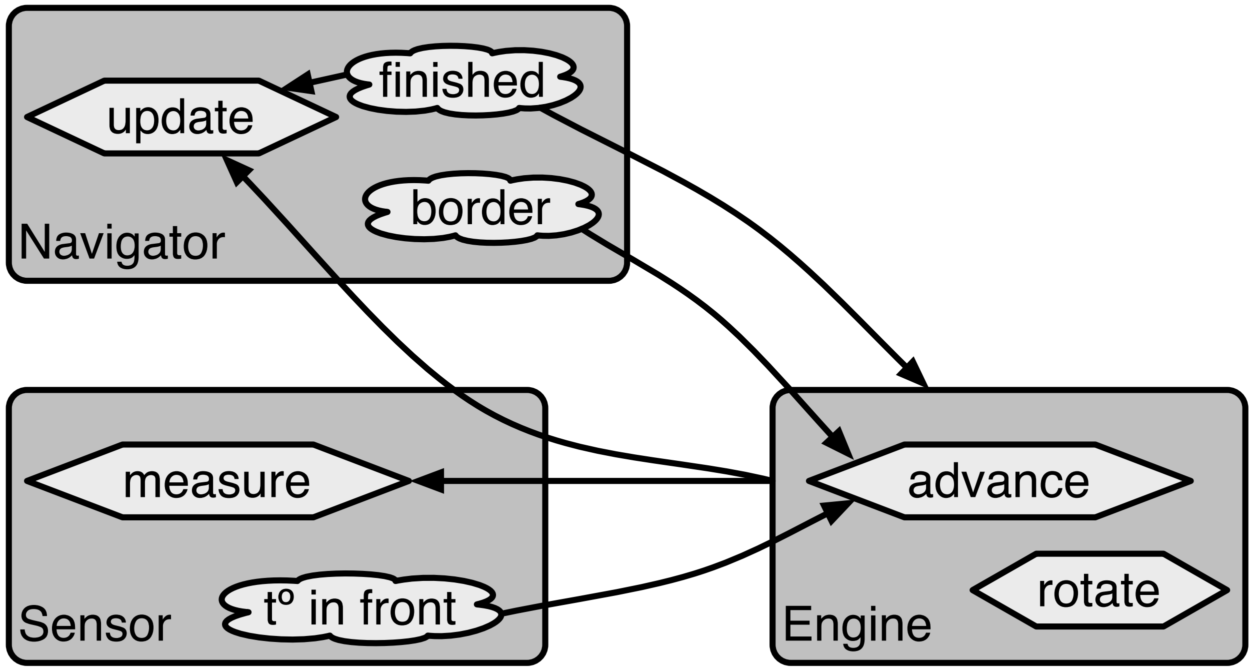
```
$ bipc.sh -I . -p HelloPackage -d "HelloCompound()" \
     --gencpp-output output
$ cd build
$ cmake ../output
$ make
$ ./build/system
```

```
package HelloPackage
  port type HelloPort_t()

  atom type HelloAtom()
    port HelloPort_t p()
    place START,END
    initial to START
    on p from START to END
  end

  compound type HelloCompound()
    component HelloAtom c1()
  end
end
```

```
[BIP ENGINE]: BIP Engine (version 2019.
[BIP ENGINE]:
[BIP ENGINE]: initialize components...
[BIP ENGINE]: random scheduling based
[BIP ENGINE]: state #0 and global time 0: 1 internal port:
[BIP ENGINE]:   [0] ROOT.c1.p [ 0, +INFTY ]
[BIP ENGINE]: -> choose [0] ROOT.c1.p at global time 8ns
[BIP ENGINE]: state #1 and global time 8ns: deadlock!
```

Example: Rescue robot





Safety constraints

Shall not advance and rotate at the same time

Shall stay within the region

Shall stay in the area that is safe or hot (but not burning)

Shall update navigation and sensor data at each move

When objective is found, the robot shall stop

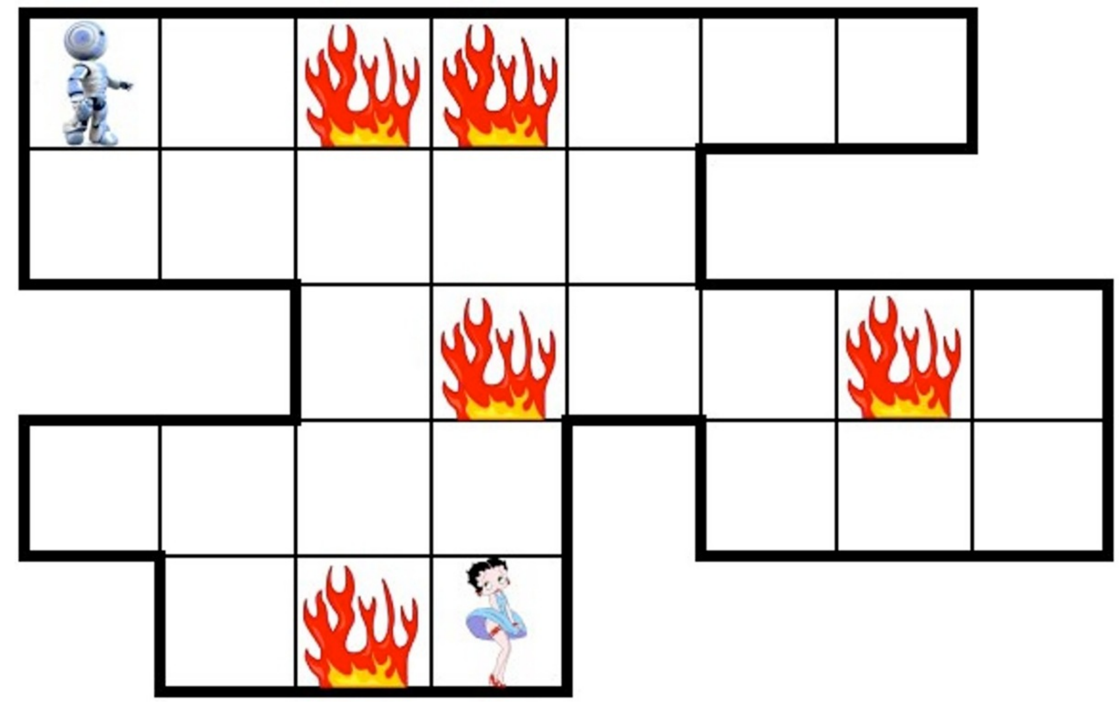
Rough plan

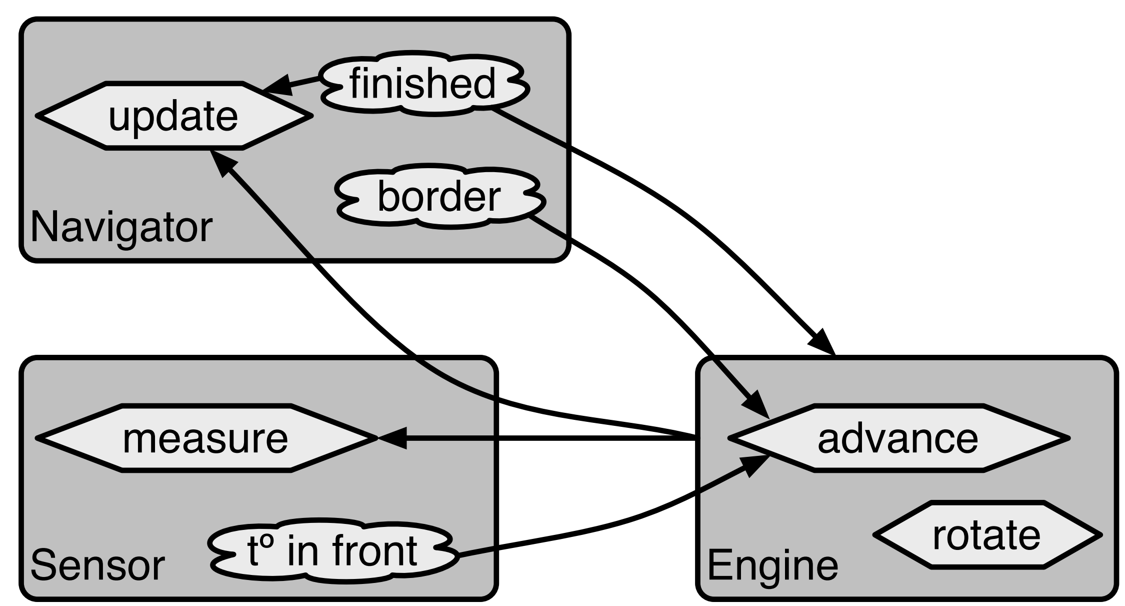
One square

N × N field (with N = 2, 5)

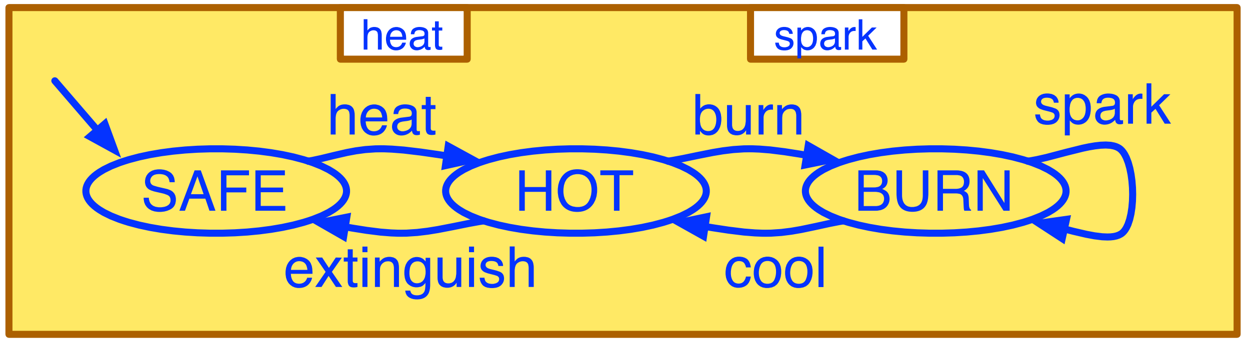
Complete with the robot

Remove the field



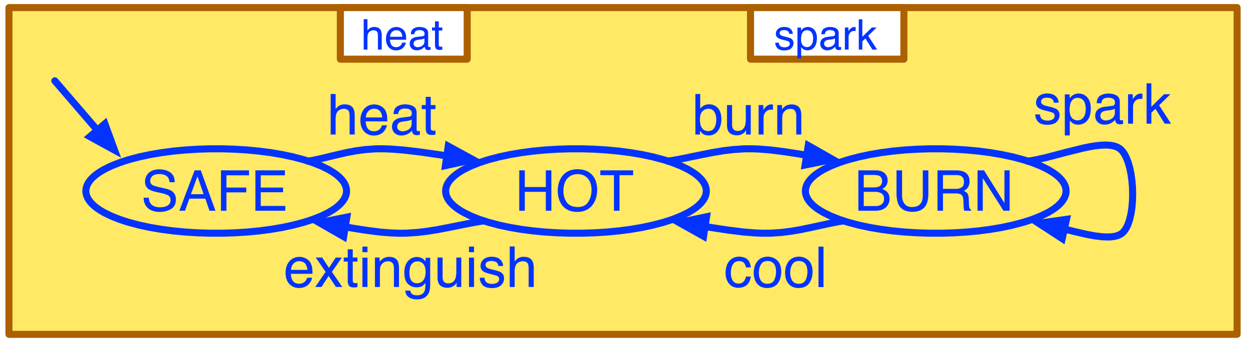


Atoms, ports and places




```
package RescueRobot
  port type Port_t()

  atom type Square()
    export port Port_t heat()
    export port Port_t spark()

    port Port_t burn()
    port Port_t cool()
    port Port_t extinguish()

    place SAFE, HOT, BURNING

    initial to SAFE
    on heat from SAFE to HOT
    on burn from HOT to BURNING
    on spark from BURNING to BURNING
    on cool from BURNING to HOT
    on extinguish from HOT to SAFE
  end

  connector type Singleton (Port_t p)
    define p
  end

  compound type Field()
    component Square square()

    connector Singleton
                  c_heat(square.heat)
    connector Singleton
                  c_spark(square.spark)
  end

  compound type RescueCompound()
    component Field field()
  end
end
```

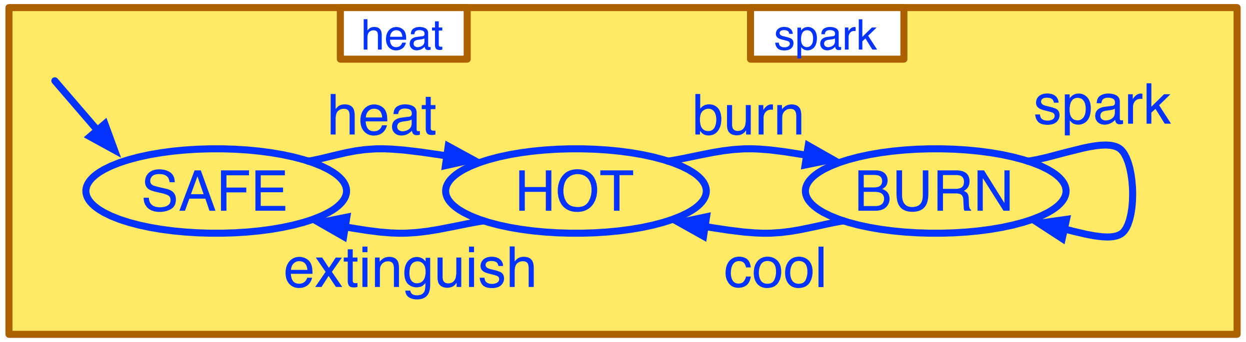
Atoms, ports and places




```
package RescueRobot
  port type Port_t()

  atom type Square()
    export port Port_t heat()
    export port Port_t spark()

    port Port_t burn()
    port Port_t cool()
    port Port_t extinguish()

    place SAFE, HOT, BURNING

    initial to SAFE
    on heat from SAFE to HOT
    on burn from HOT to BURNING
    on spark from BURNING to BURNING
    on cool from BURNING to HOT
    on extinguish from HOT to SAFE
  end
```

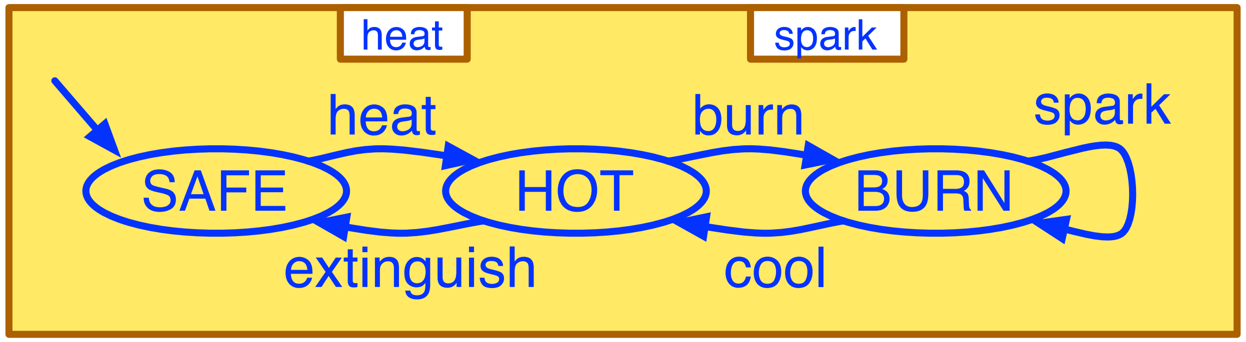
```
  connector type Singleton (Port_t p)
    define p
  end

  compound type Field()
    component Square square()

    connector Singleton
                     c_heat(square.heat)
    connector Singleton
                     c_spark(square.spark)
  end

  compound type RescueCompound()
    component Field field()
  end
end
```

Atoms, ports and places




```
package RescueRobot
  port type Port_t()

  atom type Square()
    export port Port_t heat()
    export port Port_t spark()

    port Port_t burn()
    port Port_t cool()
    port Port_t extinguish()

    place SAFE, HOT, BURNING

    initial to SAFE
    on heat from SAFE to HOT
    on burn from HOT to BURNING
    on spark from BURNING to BURNING
    on cool from BURNING to HOT
    on extinguish from HOT to SAFE
  end

    connector type Singleton (Port_t p)
      define p
    end

    compound type Field()
      component Square square()

      connector Singleton
                    c_heat(square.heat)
      connector Singleton
                    c_spark(square.spark)
    end

    compound type RescueCompound()
      component Field field()
    end
  end
```

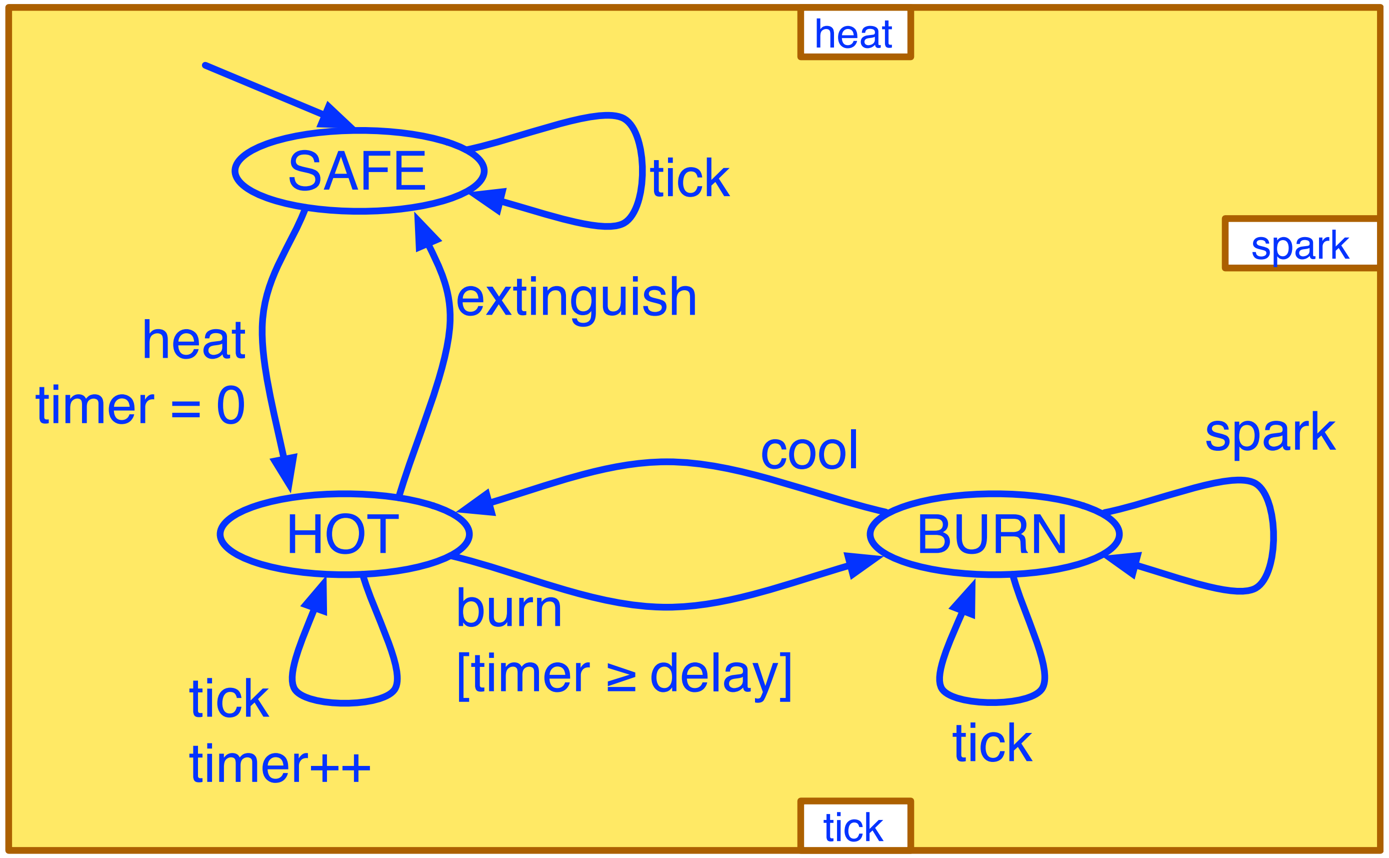
Atoms, ports and places




```
package RescueRobot
  port type Port_t()

  atom type Square()
    export port Port_t heat()
    export port Port_t spark()

    port Port_t burn()
    port Port_t cool()
    port Port_t extinguish()

    place SAFE, HOT, BURNING

    initial to SAFE
    on heat from SAFE to HOT
    on burn from HOT to BURNING
    on spark from BURNING to BURNING
    on cool from BURNING to HOT
    on extinguish from HOT to SAFE
  end
```

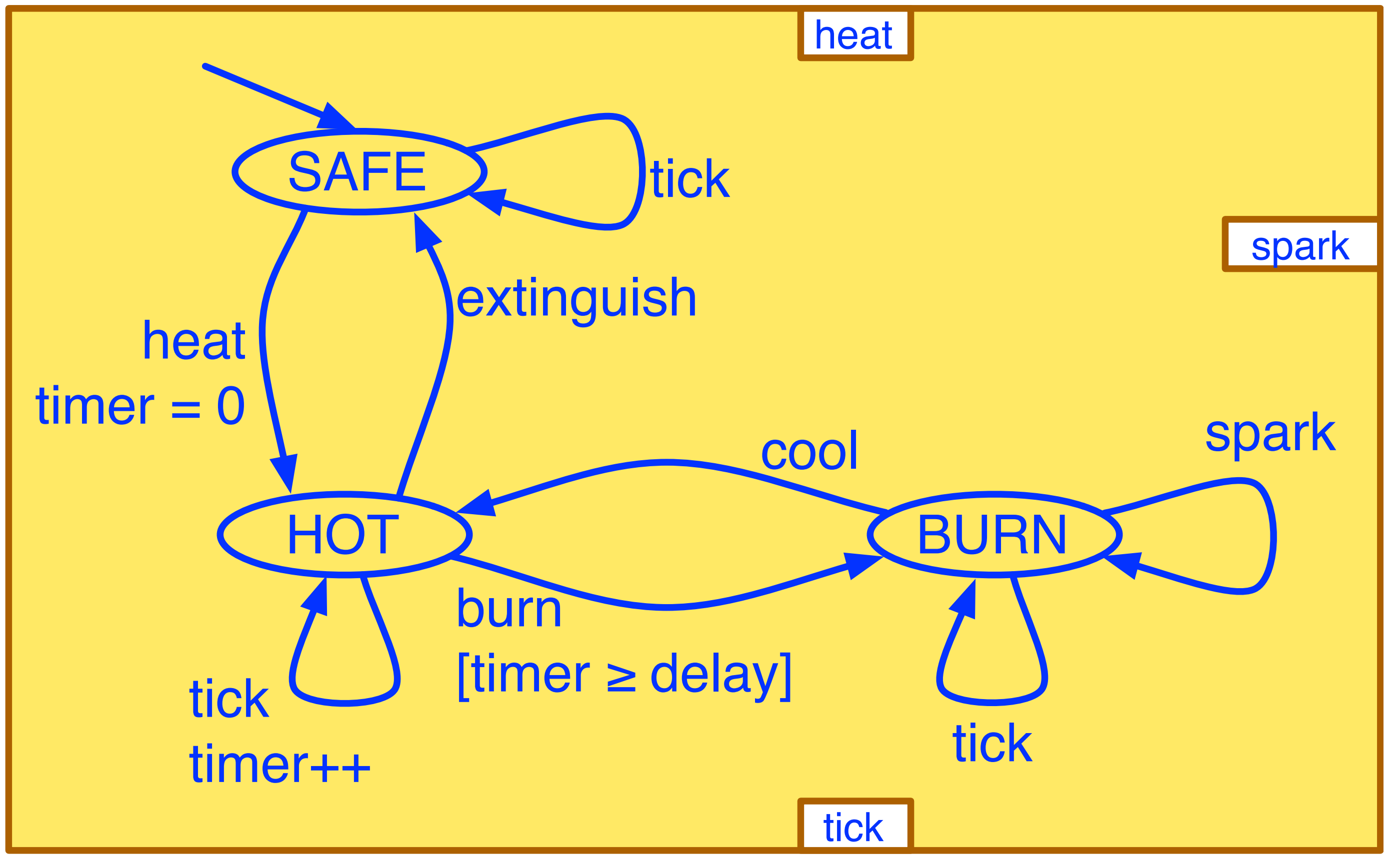
```
    connector type Singleton (Port_t p)
      define p
    end

    compound type Field()
      component Square square()

      connector Singleton
                      c_heat(square.heat)
      connector Singleton
                      c_spark(square.spark)
    end

    compound type RescueCompound()
      component Field field()
    end
  end
```

Data, guards and actions



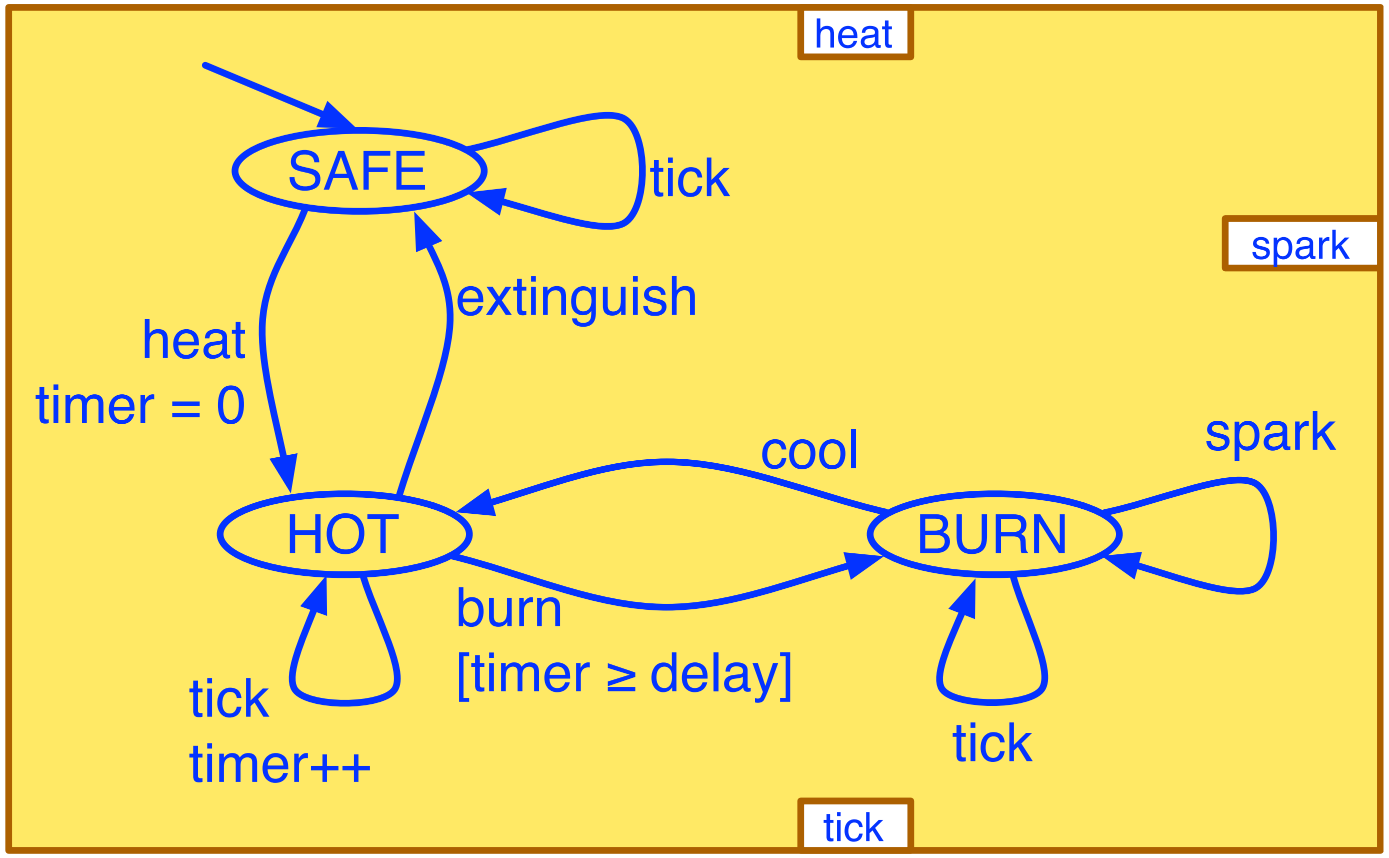

```
atom type Square (int delay)
  data int timer

  export port Port_t tick()

  <...>
  on heat from SAFE to HOT
    do {timer = 0;}

  on burn from HOT to BURNING
    provided (timer >= delay)

  on cool from BURNING to HOT
    do {timer = 0;}
  <...>

  on tick from SAFE to SAFE
  on tick from HOT to HOT
    do {timer = timer + 1;}
  on tick from BURNING to BURNING
end
```

Data, guards and actions



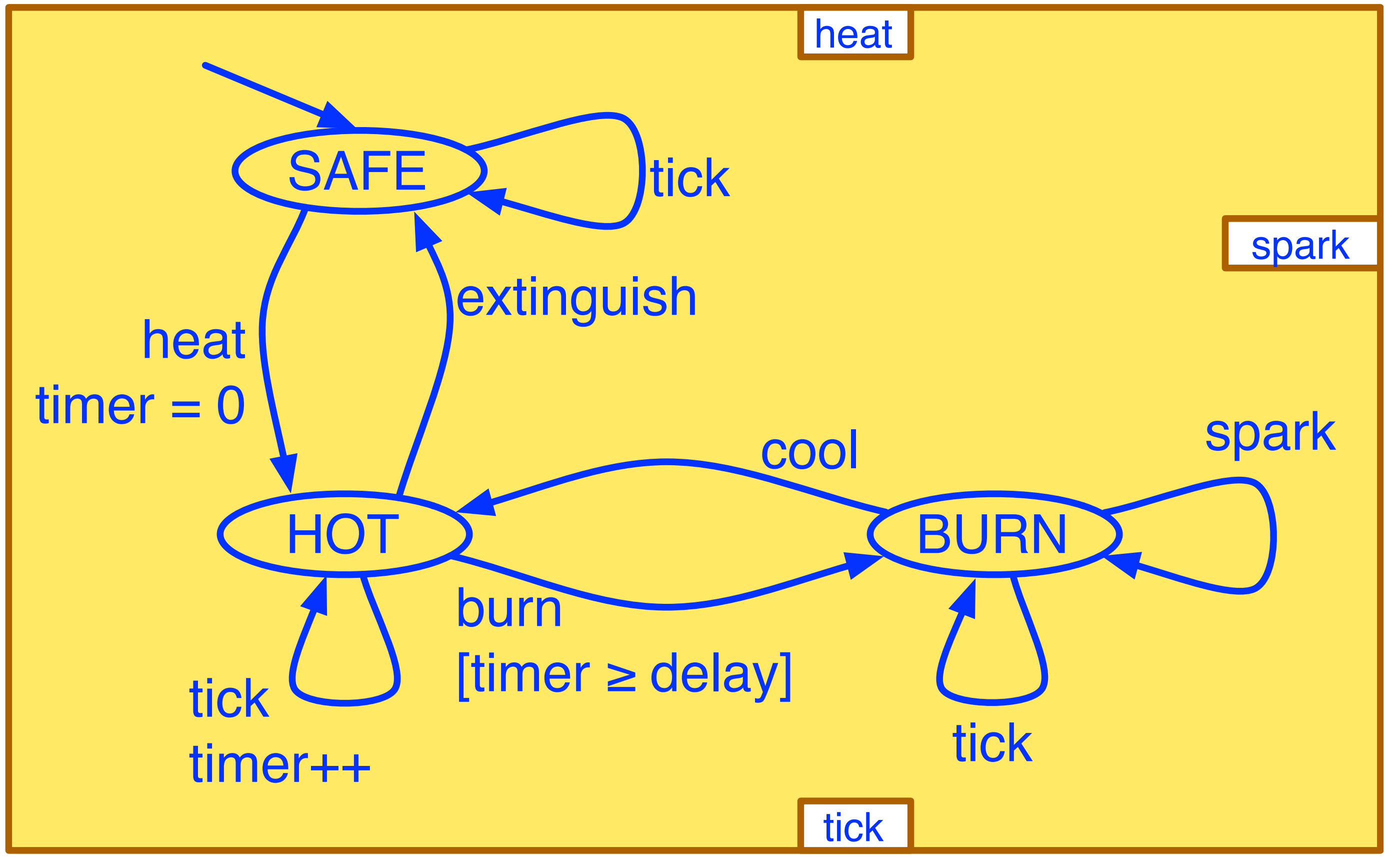

```
atom type Square (int delay)
  data int timer

  export port Port_t tick()

  <...>
  on heat from SAFE to HOT
    do {timer = 0;}

  on burn from HOT to BURNING
    provided (timer >= delay)

  on cool from BURNING to HOT
    do {timer = 0;}
  <...>

  on tick from SAFE to SAFE
  on tick from HOT to HOT
    do {timer = timer + 1;}
  on tick from BURNING to BURNING
end
```

Data, guards and actions



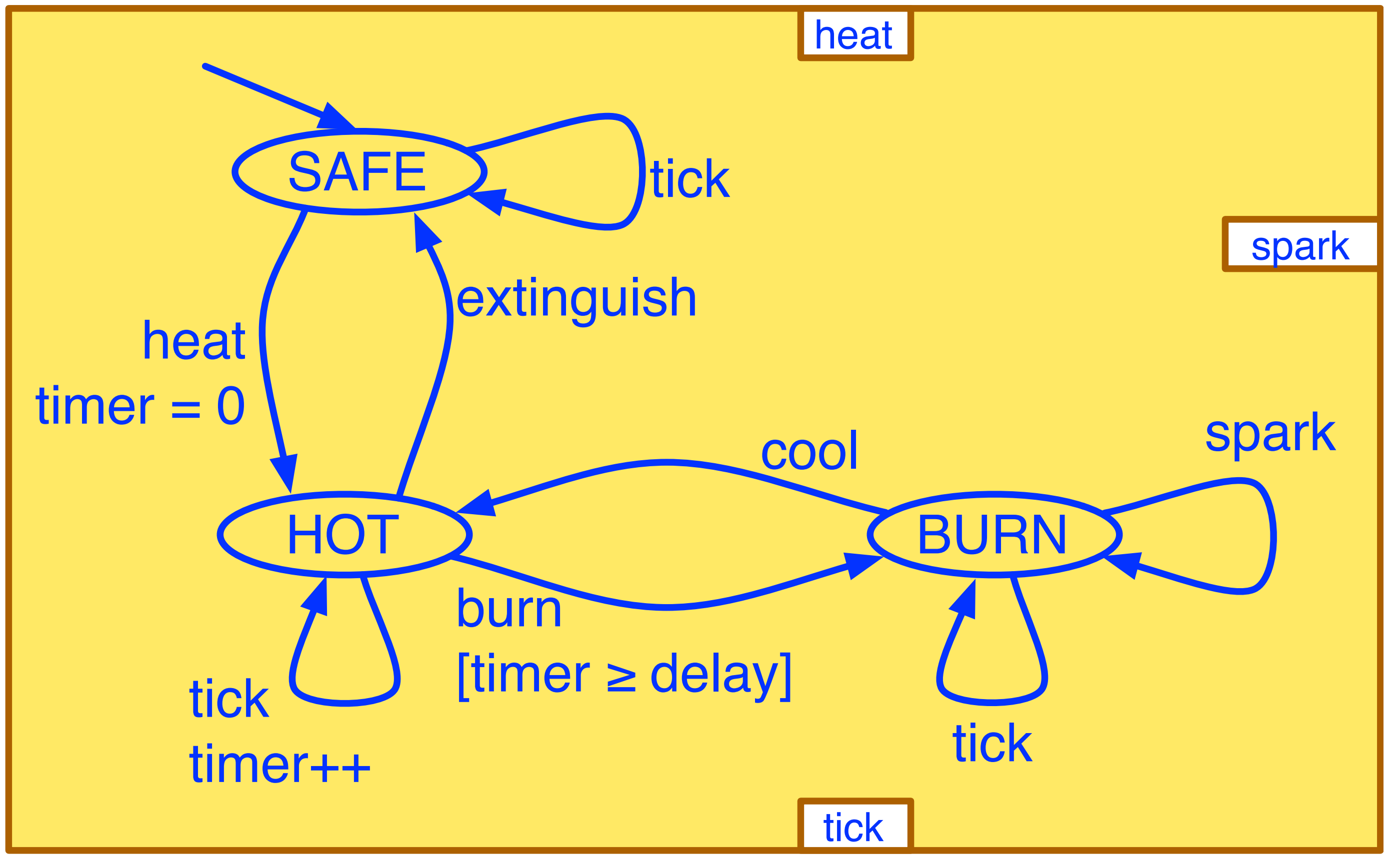

```
atom type Square (int delay)
  data int timer

  export port Port_t tick()

  <...>
  on heat from SAFE to HOT
    do {timer = 0;}

  on burn from HOT to BURNING
    provided (timer >= delay)

  on cool from BURNING to HOT
    do {timer = 0;}
  <...>

  on tick from SAFE to SAFE
  on tick from HOT to HOT
    do {timer = timer + 1;}
  on tick from BURNING to BURNING
end
```

Data, guards and actions



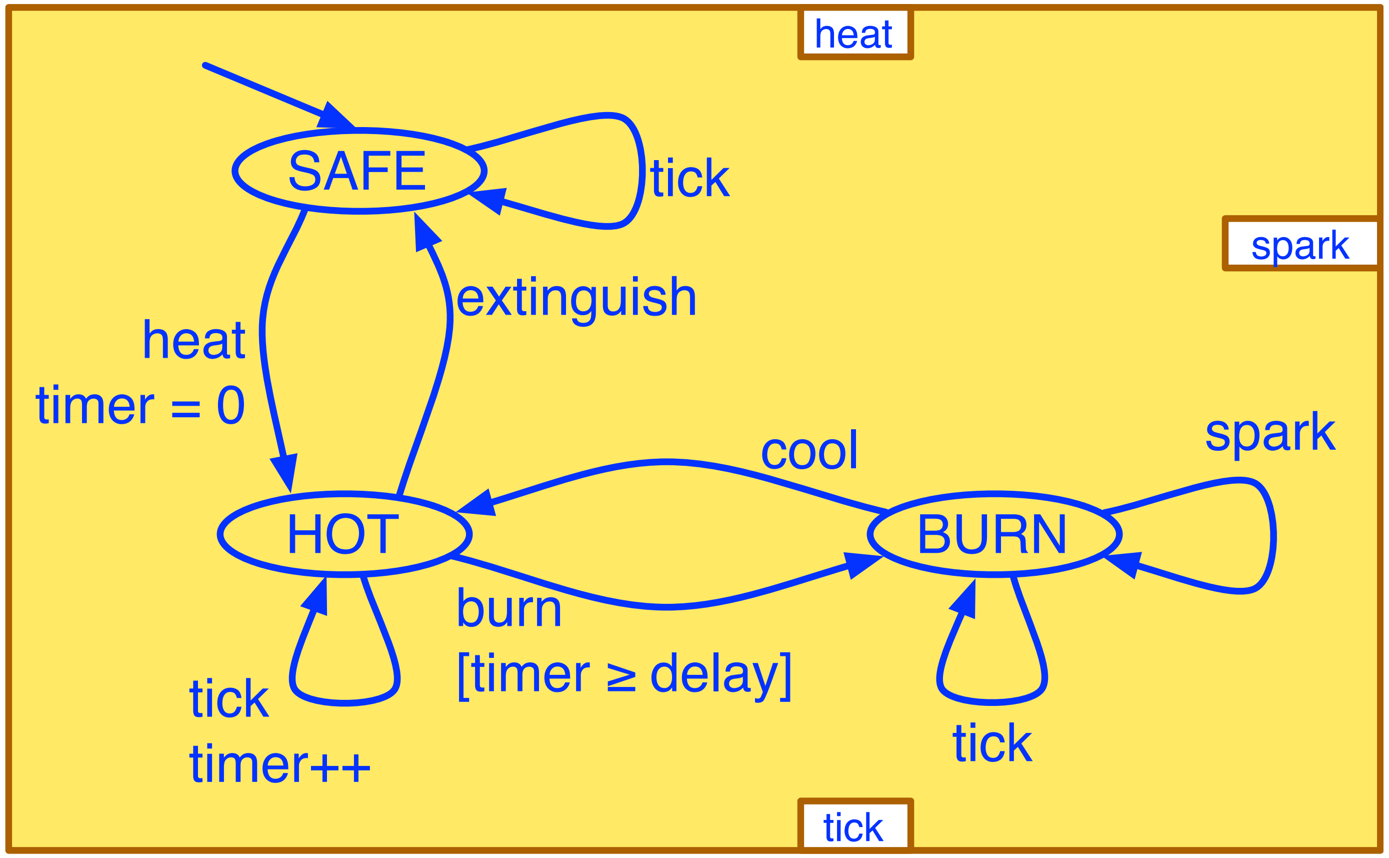

```
atom type Square (int delay)
  data int timer

  export port Port_t tick()

  <...>
  on heat from SAFE to HOT
    do {timer = 0;}
  on burn from HOT to BURNING
    provided (timer >= delay)

  on cool from BURNING to HOT
    do {timer = 0;}
  <...>

  on tick from SAFE to SAFE
  on tick from HOT to HOT
    do {timer = timer + 1;}
  on tick from BURNING to BURNING
end
```

Data, guards and actions



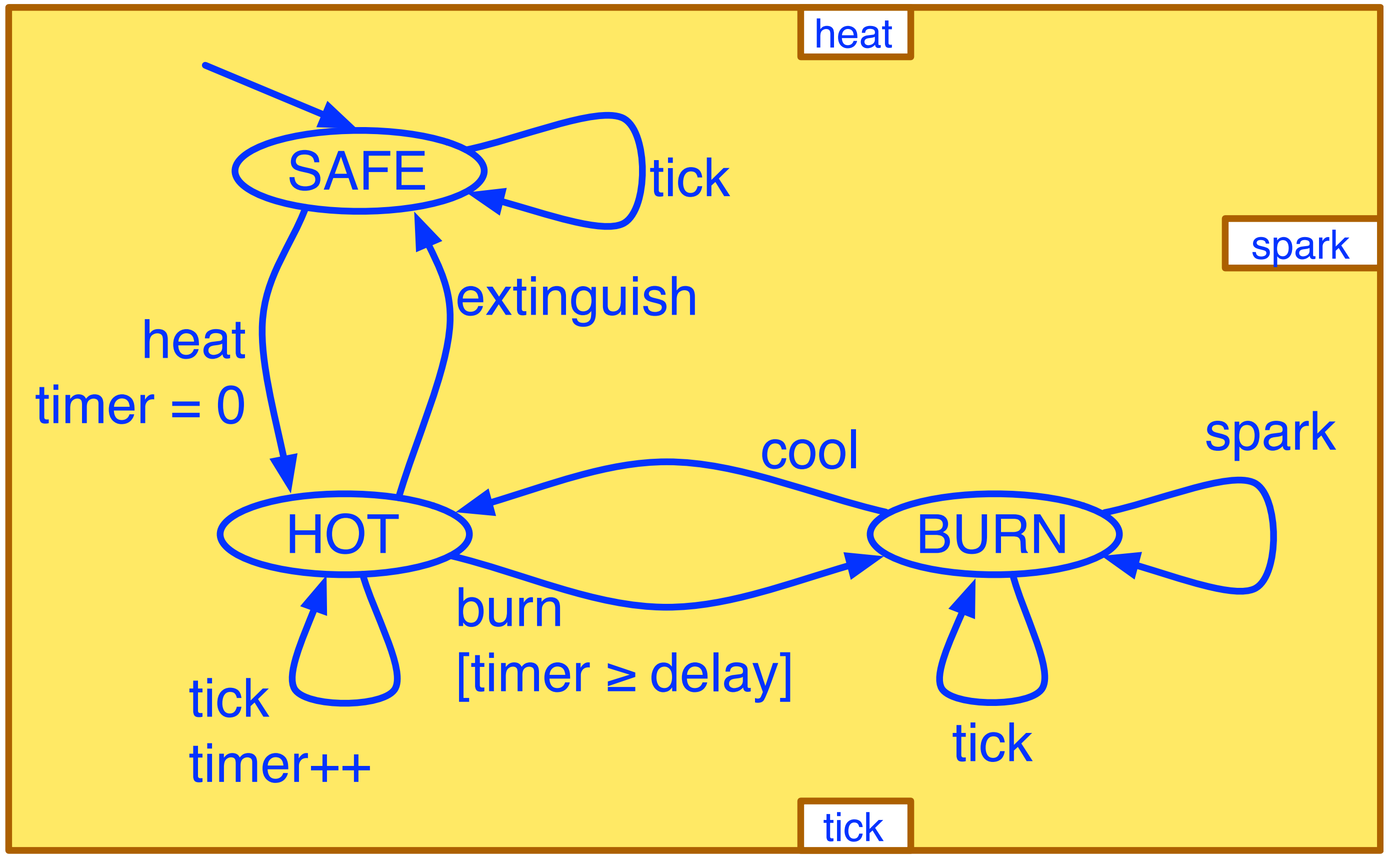

```
atom type Square (int delay)
  data int timer

  export port Port_t tick()

  <...>
  on heat from SAFE to HOT
    do {timer = 0;}
  on burn from HOT to BURNING
    provided (timer >= delay)

  on cool from BURNING to HOT
    do {timer = 0;}
  <...>

  on tick from SAFE to SAFE
  on tick from HOT to HOT
    do {timer = timer + 1;}
  on tick from BURNING to BURNING
end
```

Data, guards and actions




```
atom type Square (int delay)
  data int timer

  export port Port_t tick()

  <...>
  on heat from SAFE to HOT
    do {timer = 0;}
  on burn from HOT to BURNING
    provided (timer >= delay)

  on cool from BURNING to HOT
    do {timer = 0;}
  <...>

  on tick from SAFE to SAFE
  on tick from HOT to HOT
    do {timer = timer + 1;}
  on tick from BURNING to BURNING
end
```

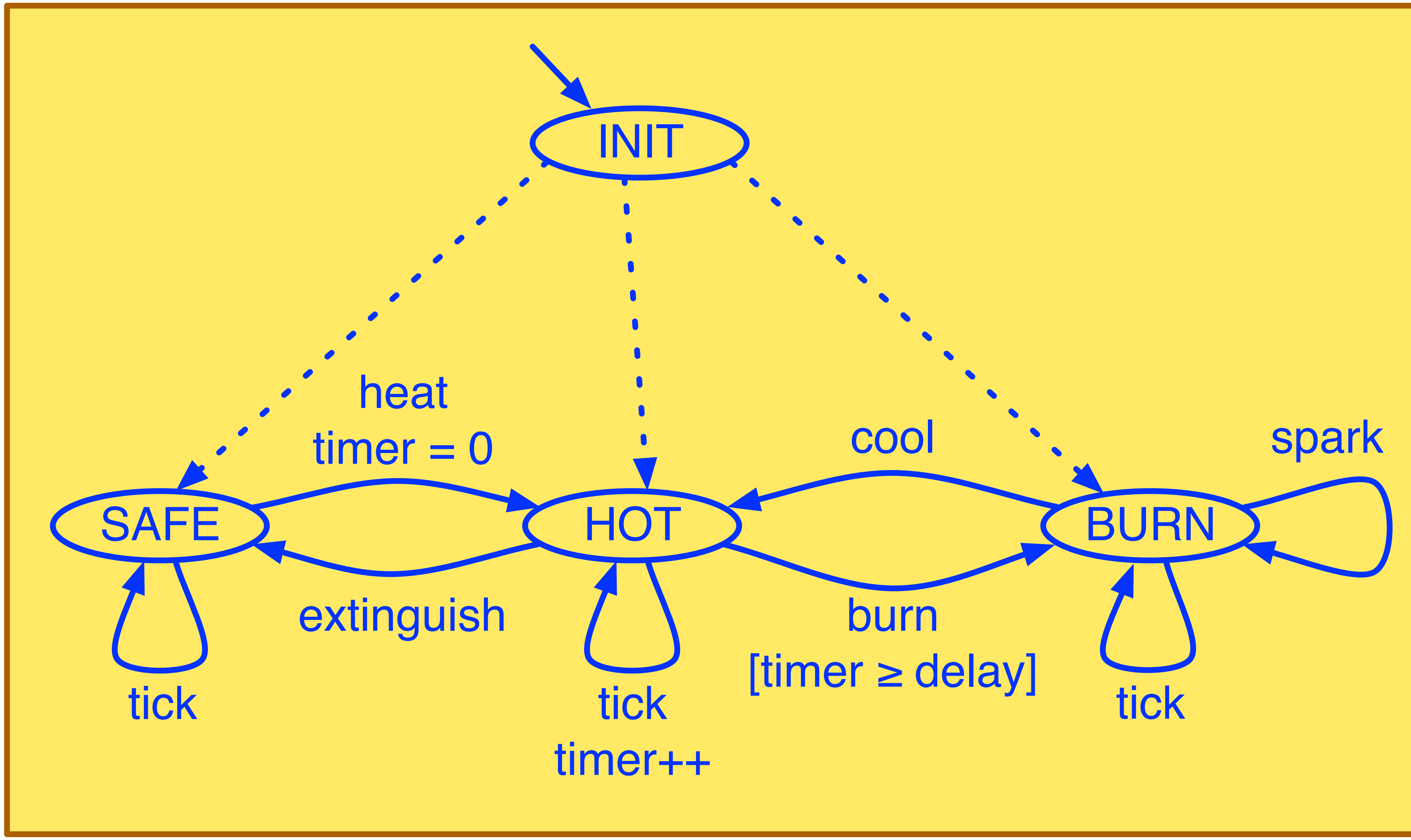
Data, guards and actions




```
atom type Square (int delay)
  data int timer

  export port Port_t tick()

  <...>
  on heat from SAFE to HOT
    do {timer = 0;}
  on burn from HOT to BURNING
    provided (timer >= delay)

  on cool from BURNING to HOT
    do {timer = 0;}
  <...>

  on tick from SAFE to SAFE
  on tick from HOT to HOT
    do {timer = timer + 1;}
  on tick from BURNING to BURNING
end
```

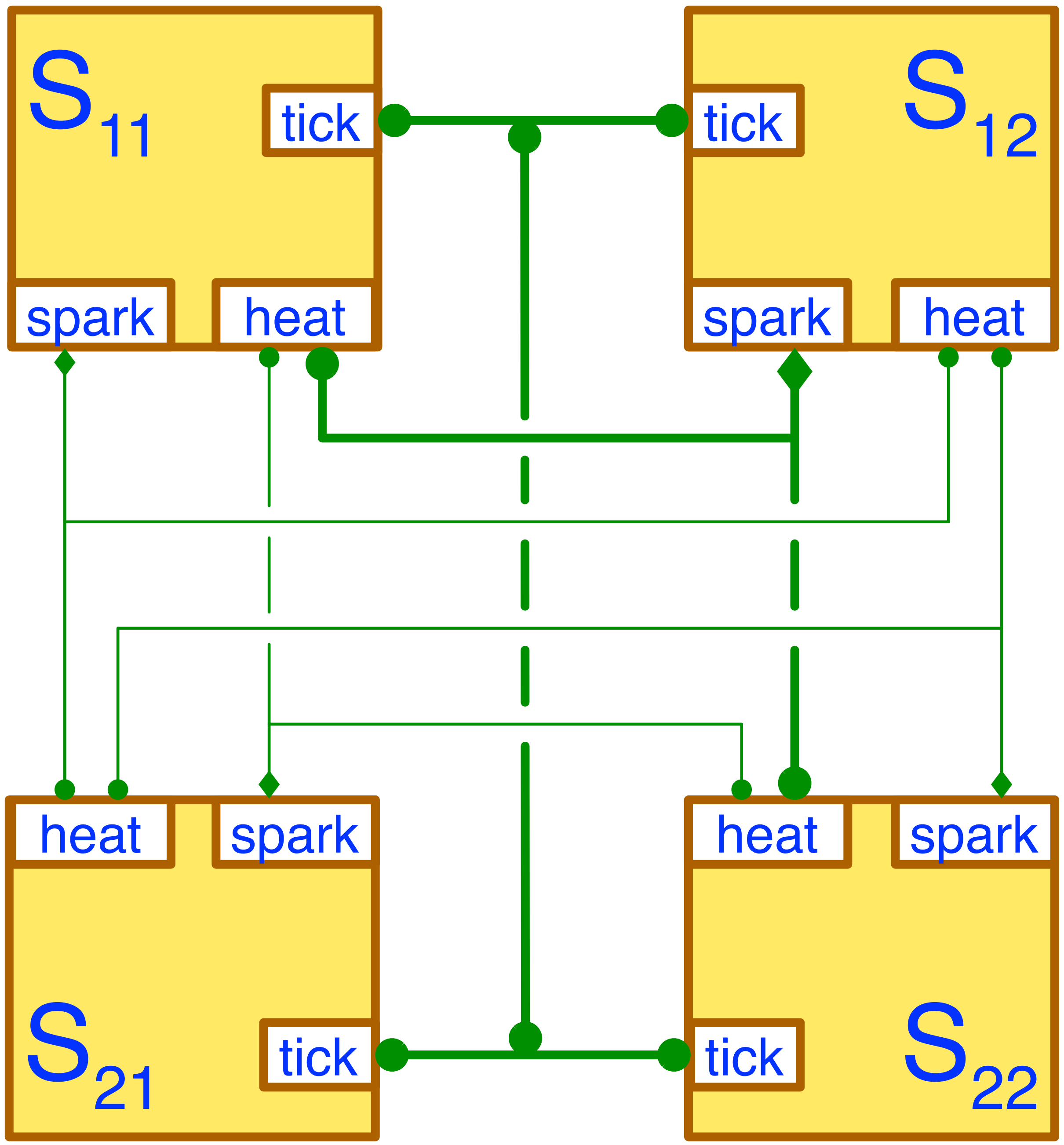
1. Add volatility
2. Add initial temperature

Internal transitions



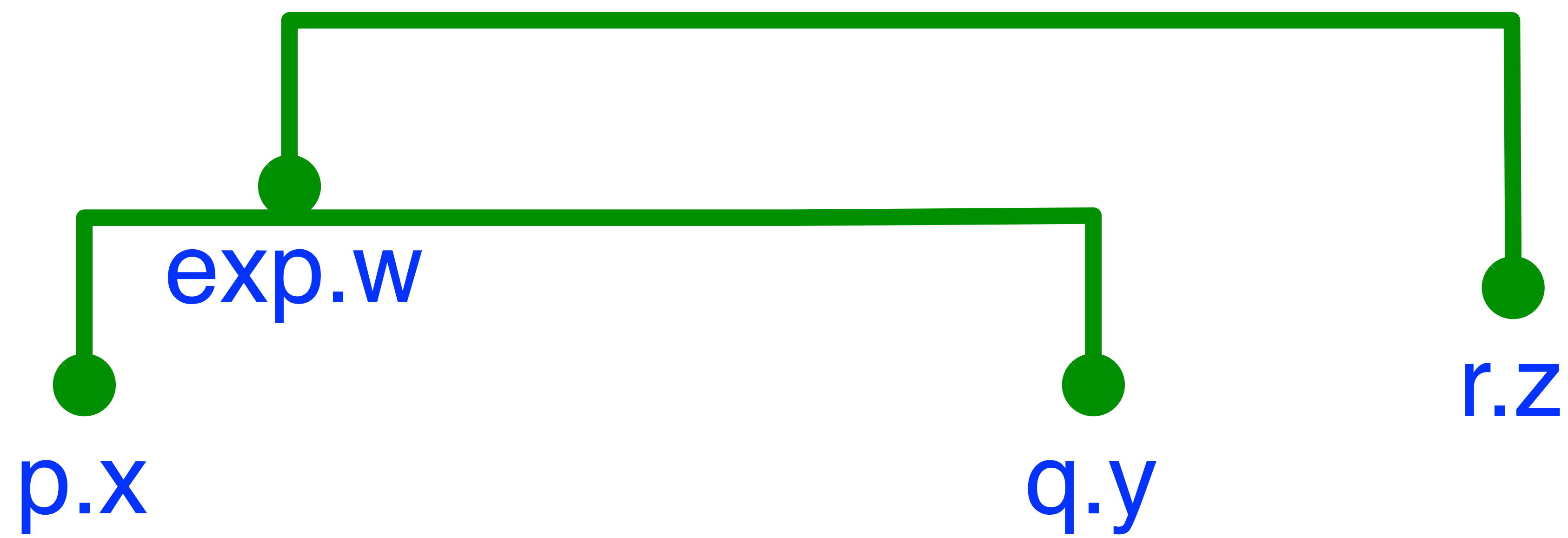
internal from INIT **to ...**

Connectors



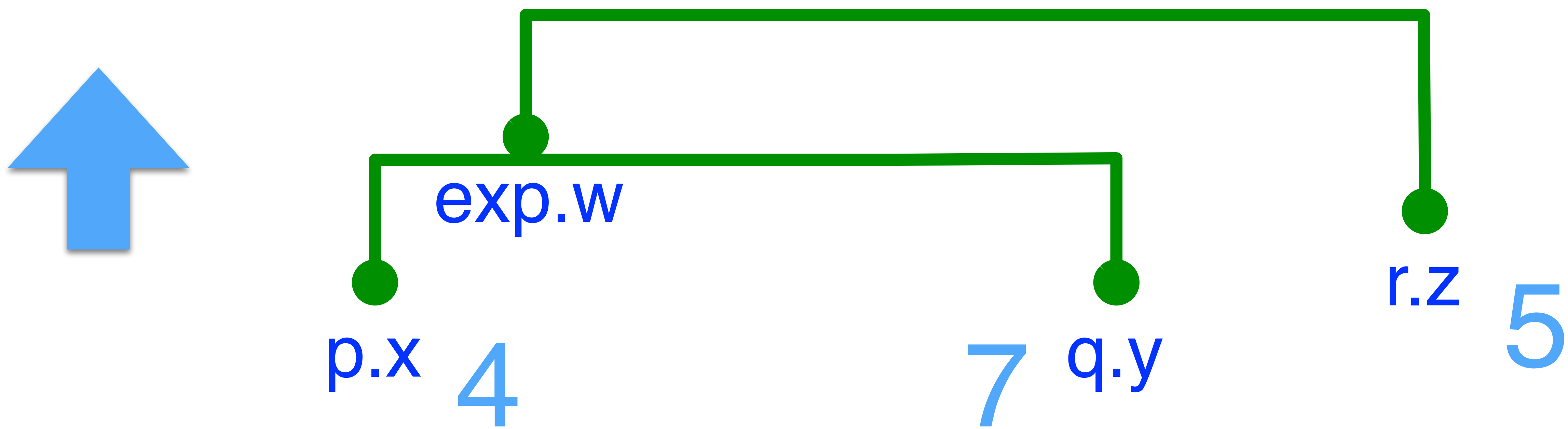

```
connector type Synchron2 (
        Port_t p, Port_t
    )
    export port Port_t sync_port()
    define p q
end

connector type Trigger2 (
        Port_t p, Port_t q, Port_t r
    )
    define p' q r
end

<...>

connector Synchron2 c_tick1 (
        square11.tick, square12.tick
)
connector Synchron2 c_tick2 (
        square21.tick, square22.tick
)

connector Synchron2 c_tick (
  c_tick1.sync_port, c_tick2.sync_port
)
```

Data transfer



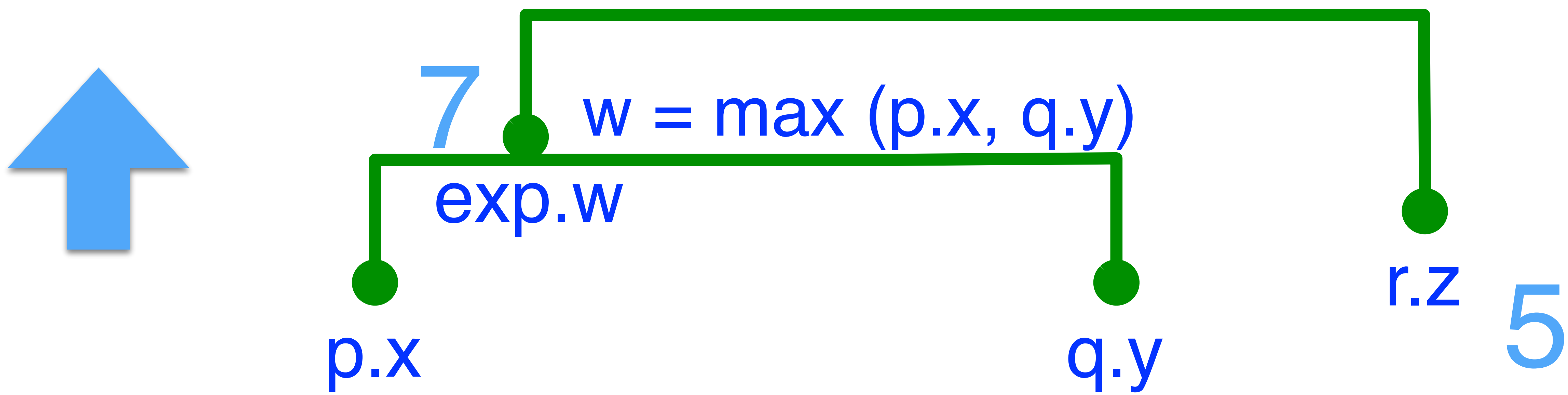

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer



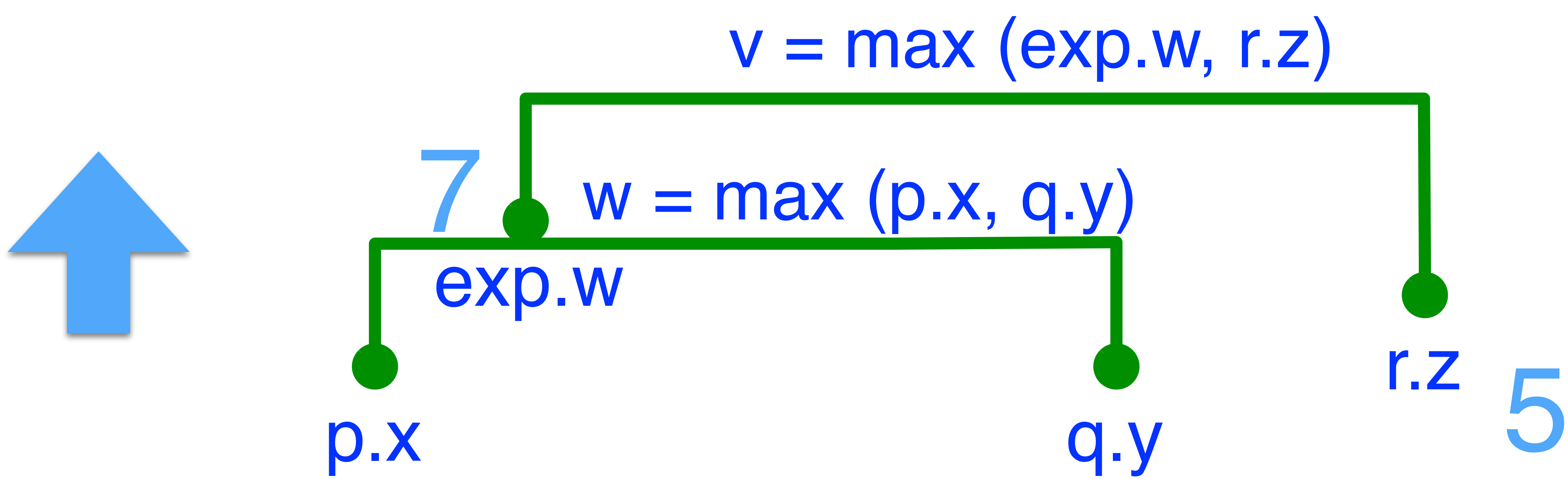

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer



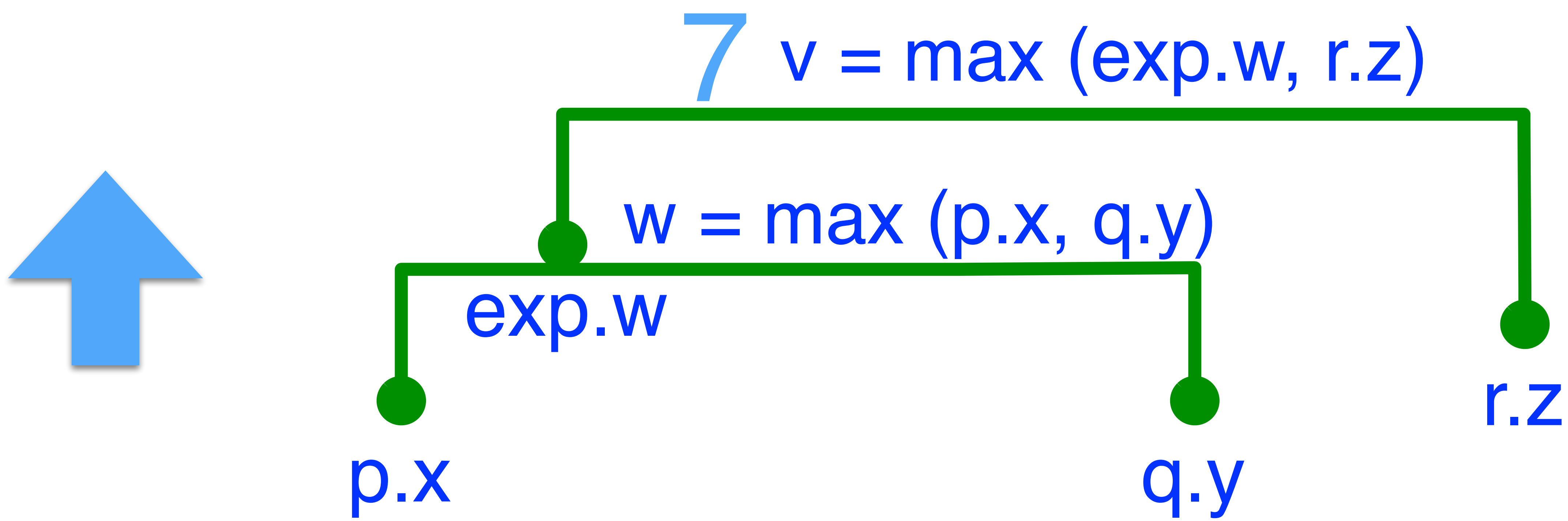

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer



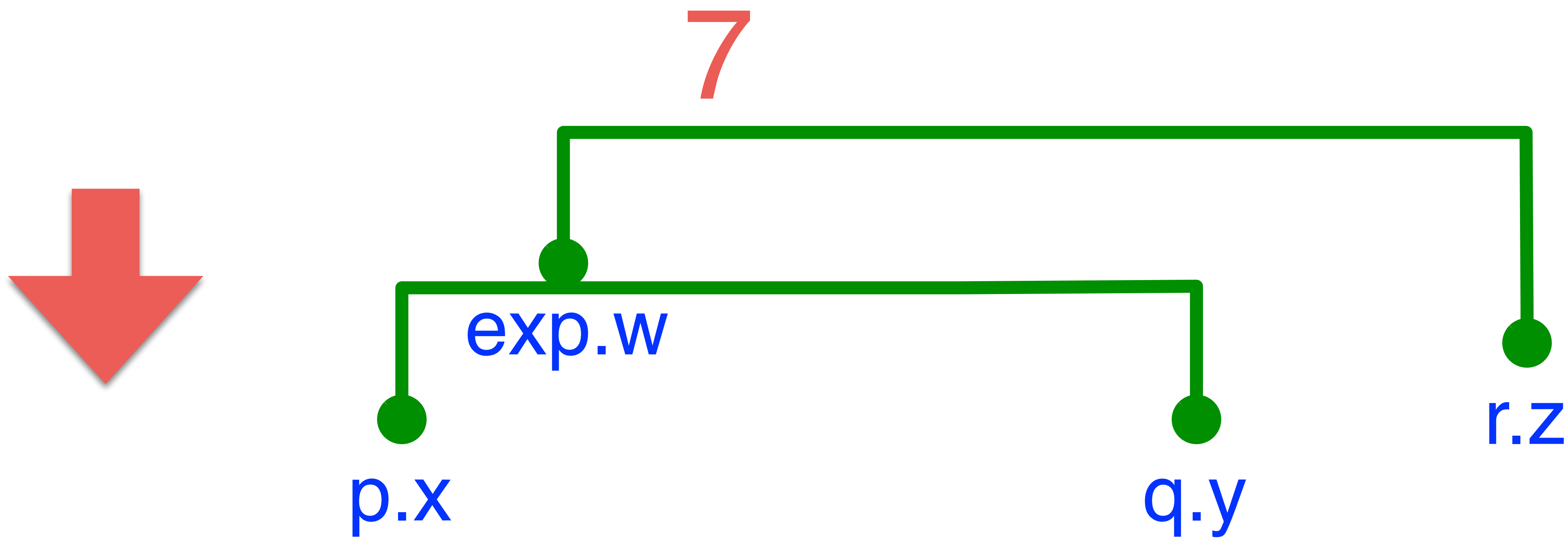

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer



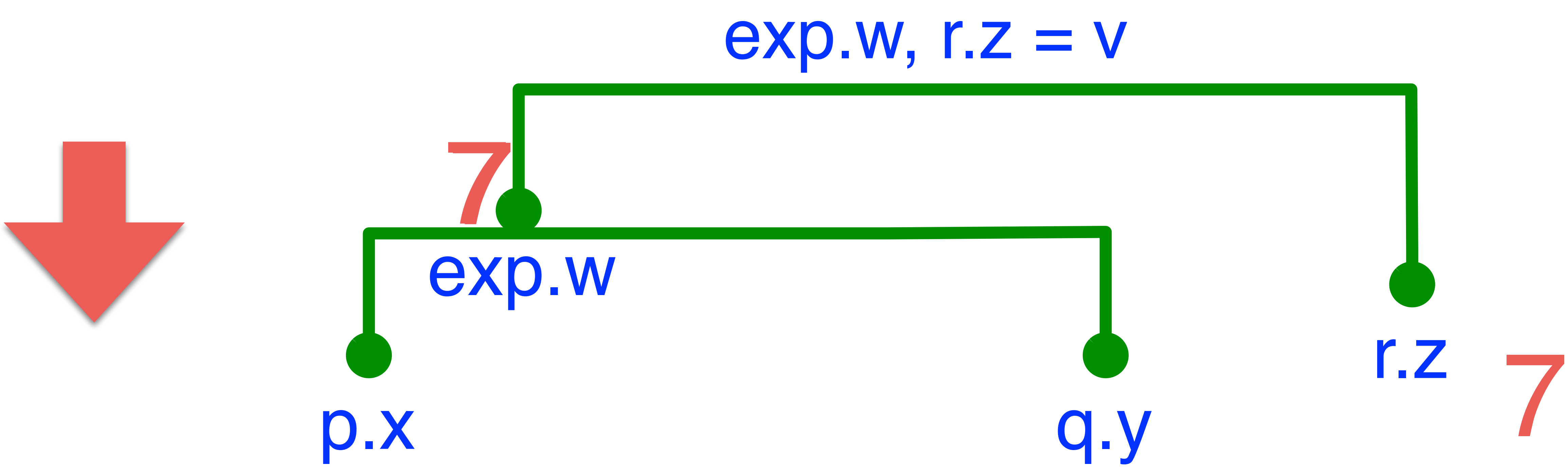

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer



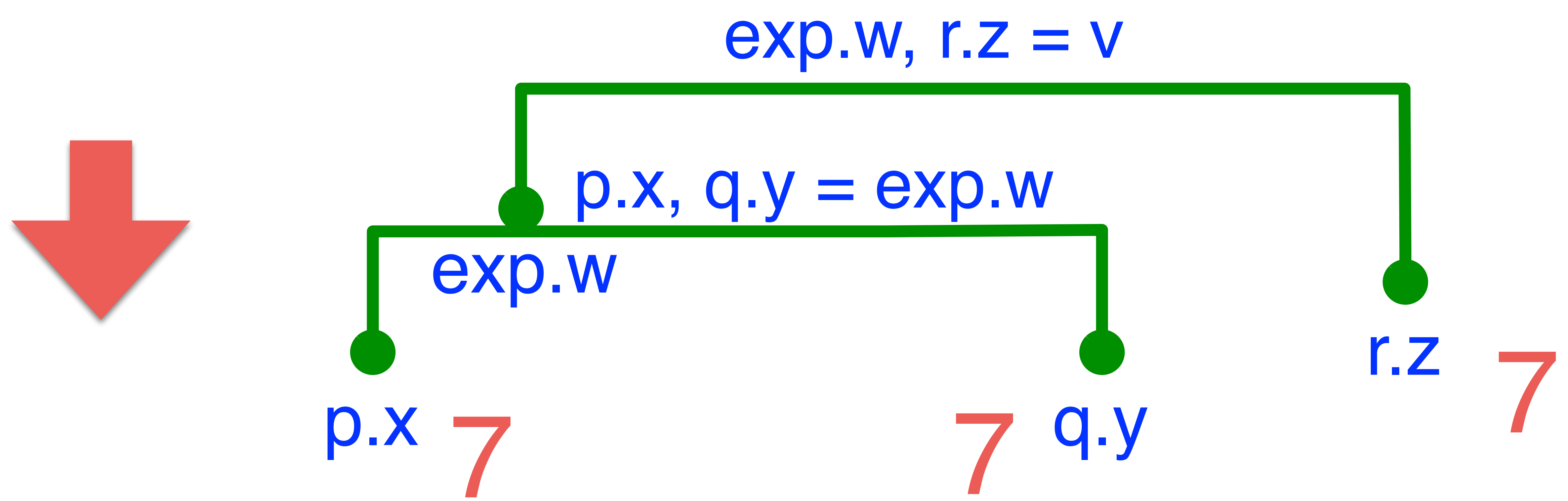

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer



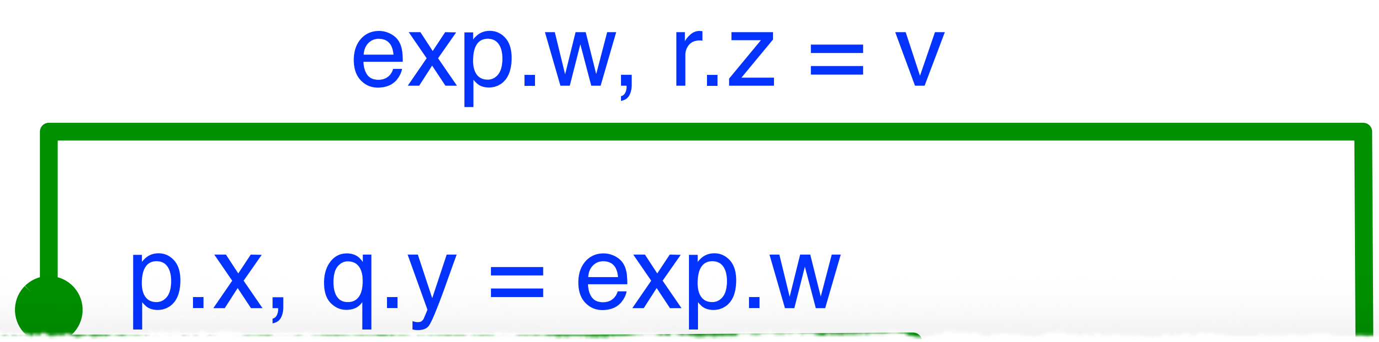

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer




```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Data transfer



1. Add connectors to gather and print information about the temperature in all squares of the field.
2. Add an atom to enforce this after each tick of the clock.

7

```
connector type Max (Port_int p, Port_int q)
   data int w
   export port Port_int exp(w)
   define p q
   up {w = max(p.v, q.v);}
   down {p.v = w; q.v = w;}
end
```

Components of the robot

Safety constraints

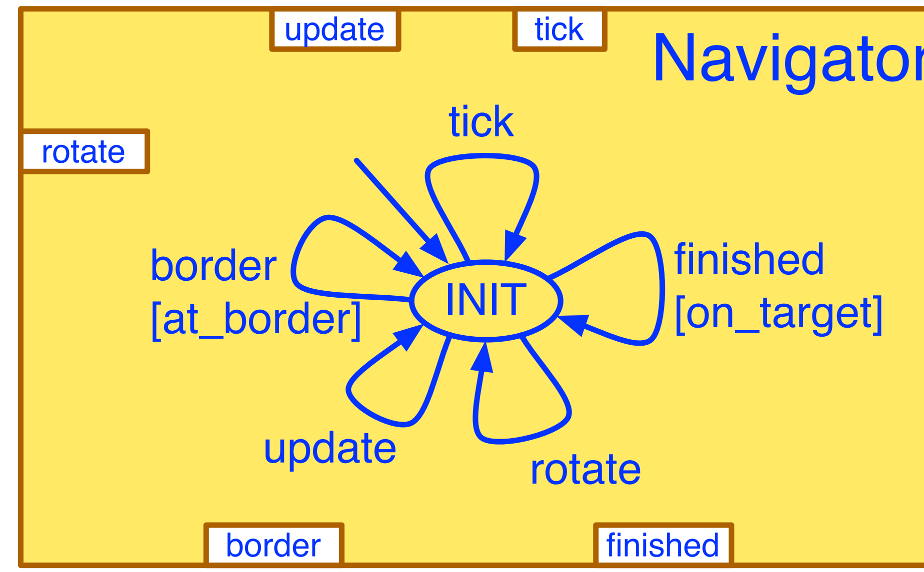
Shall not advance and rotate at the same time

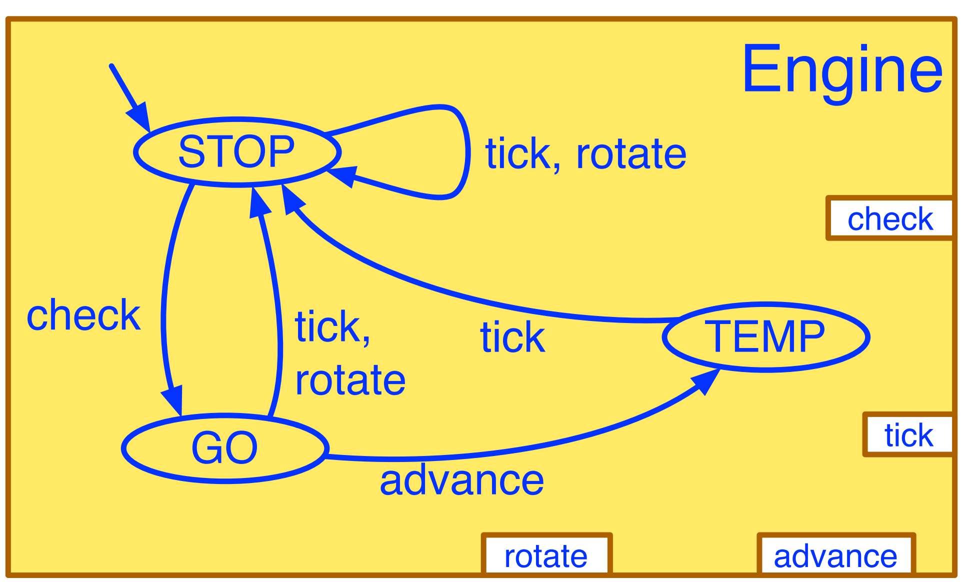
Shall stay within the region

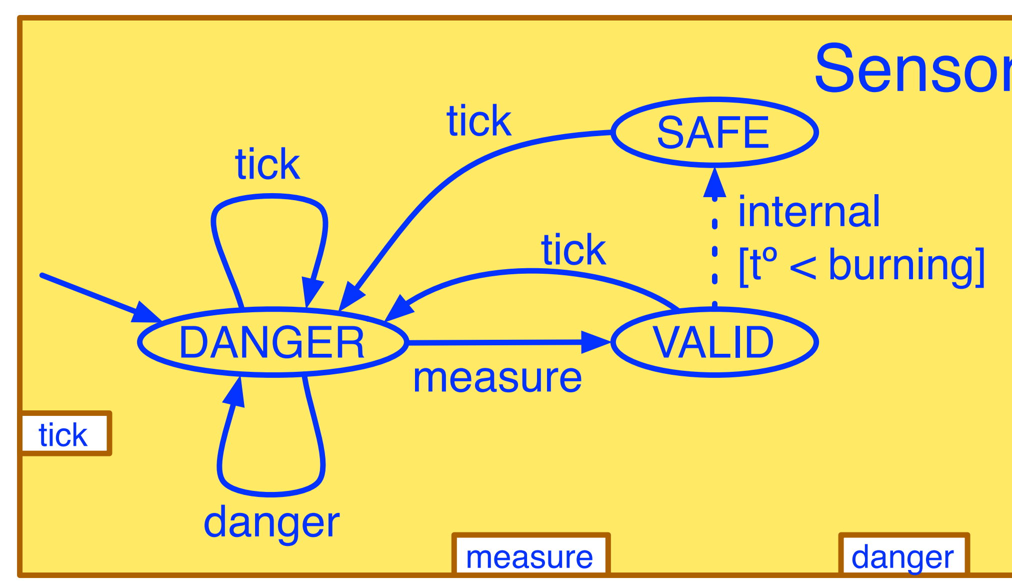
Shall stay in the area that is safe or hot (but not burning)

Shall update navigation and sensor data at each move

When objective is found, the robot shall stop



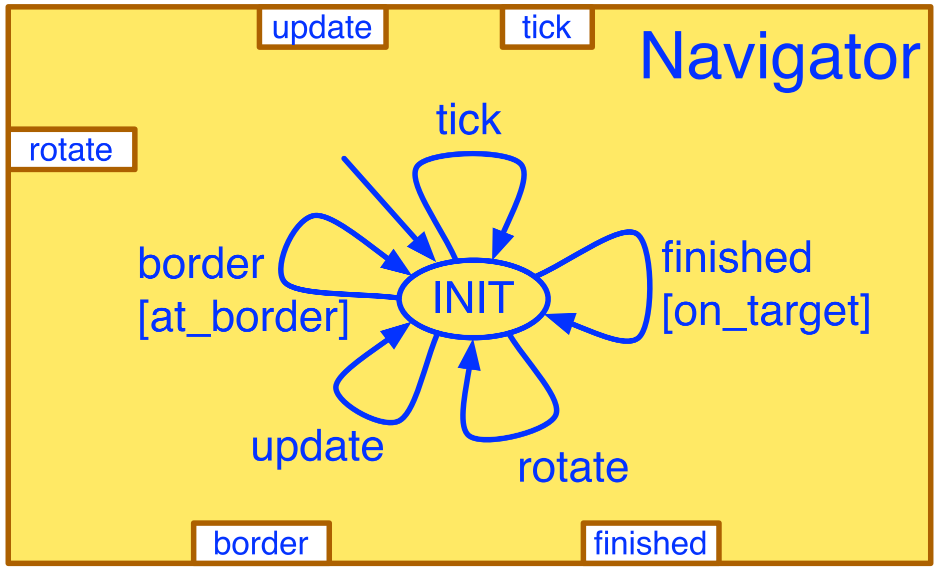


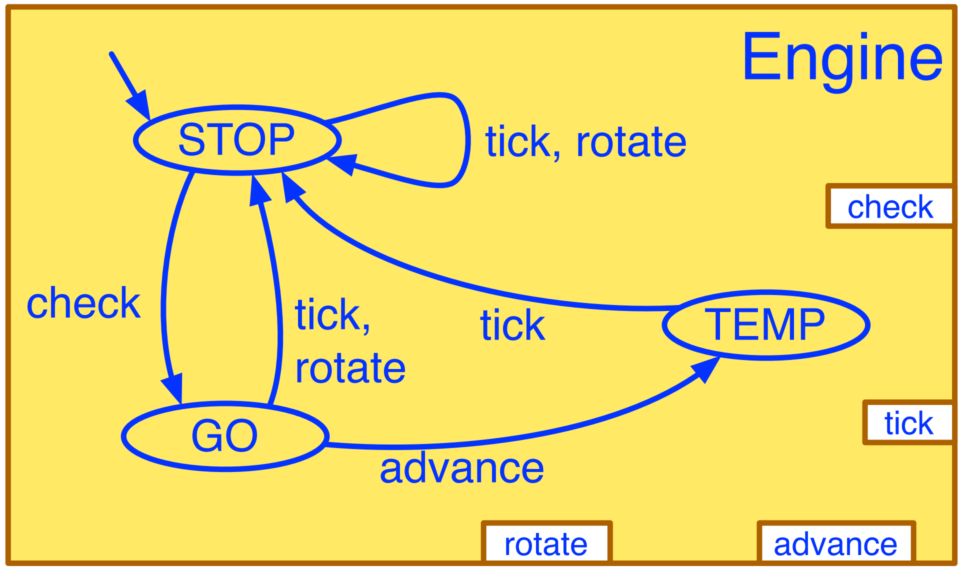


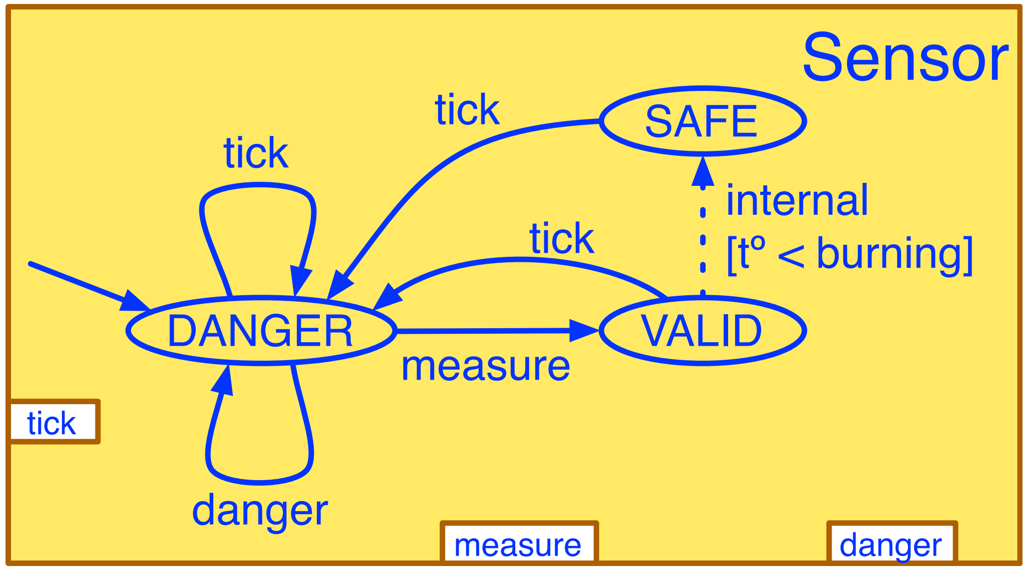
Components of the robot

Safety constraints

- Shall not advance and rotate at the same time
- Shall stay within the region
- Shall stay in the area that is safe or hot (but not burning)
- Shall update navigation and sensor data at each move
- When objective is found, the robot shall stop



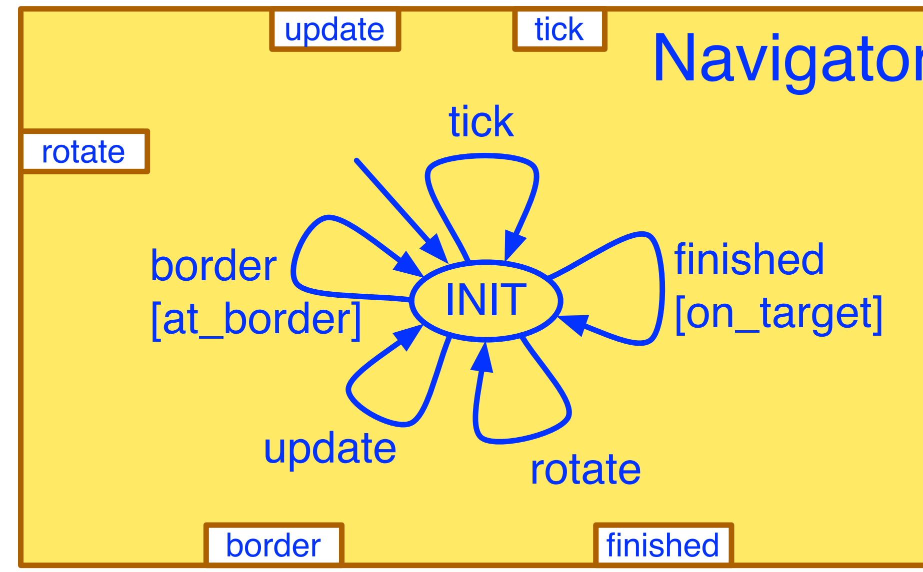


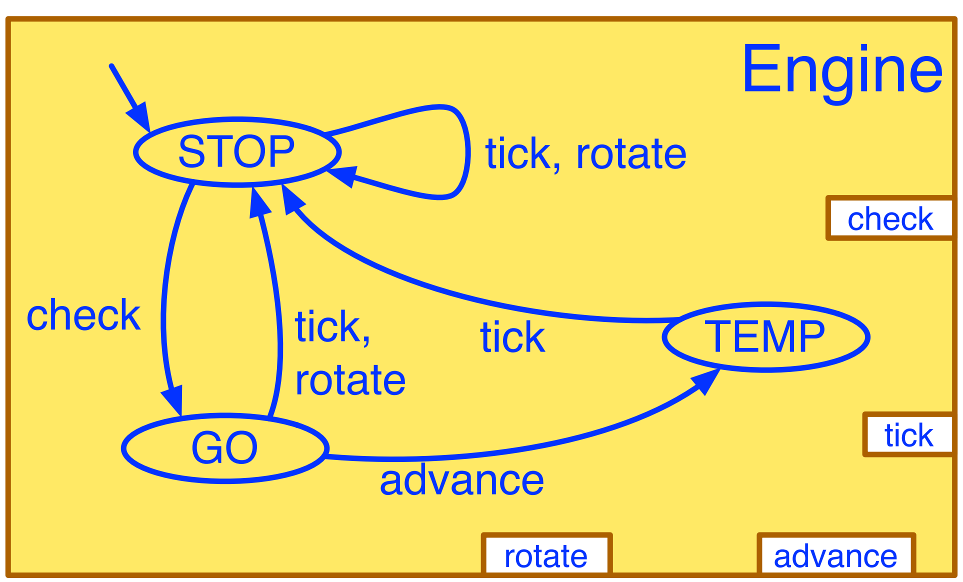


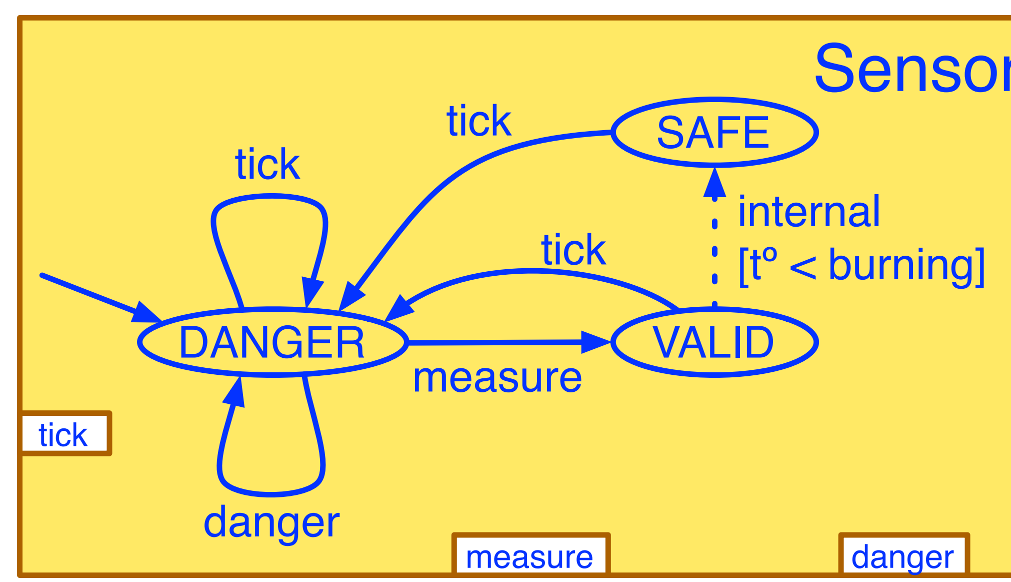
Components of the robot

Safety constraints

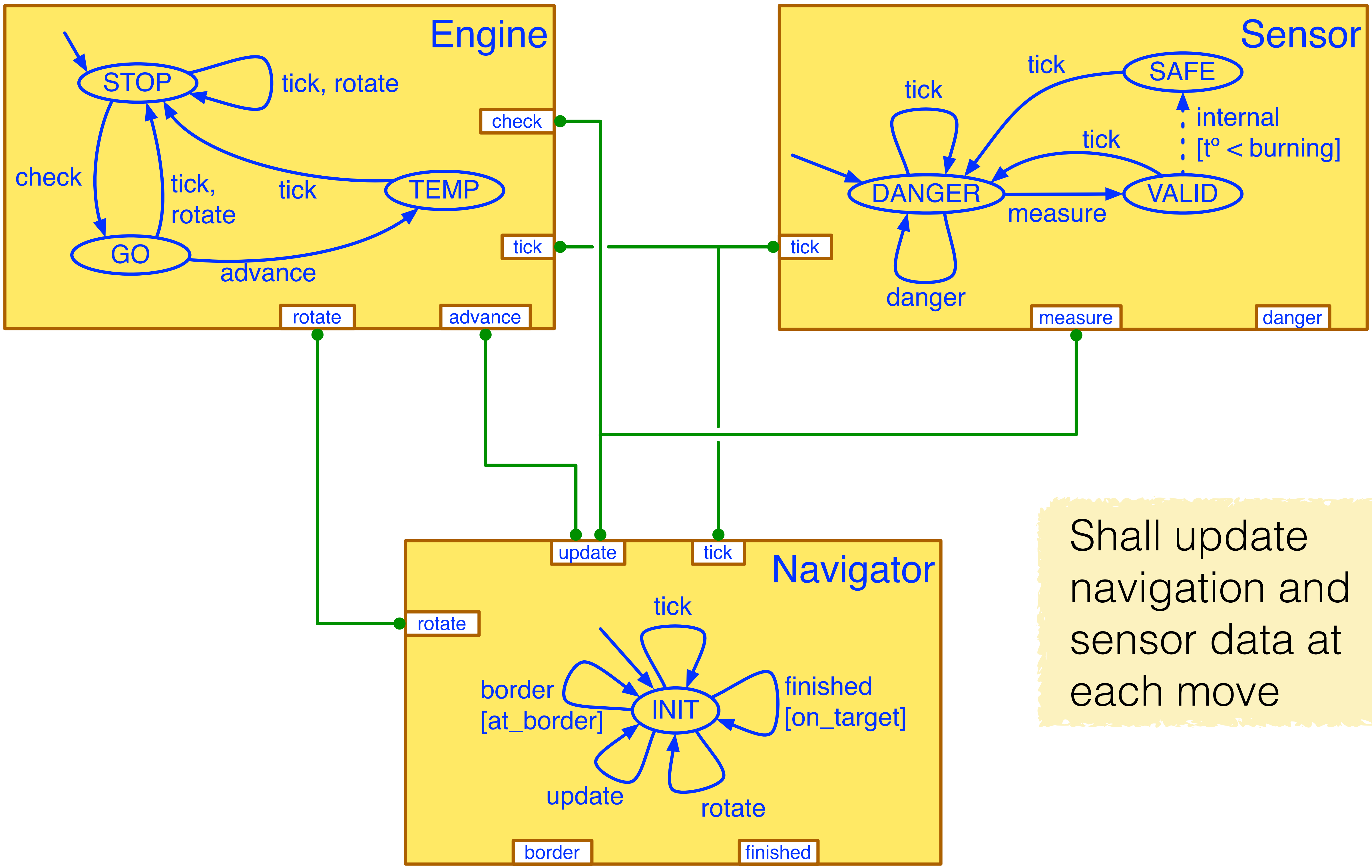
- ✓ Shall not advance and rotate at the same time
- Shall stay within the region
- Shall stay in the area that is safe or hot (but not burning)
- Shall update navigation and sensor data at each move
- When objective is found, the robot shall stop





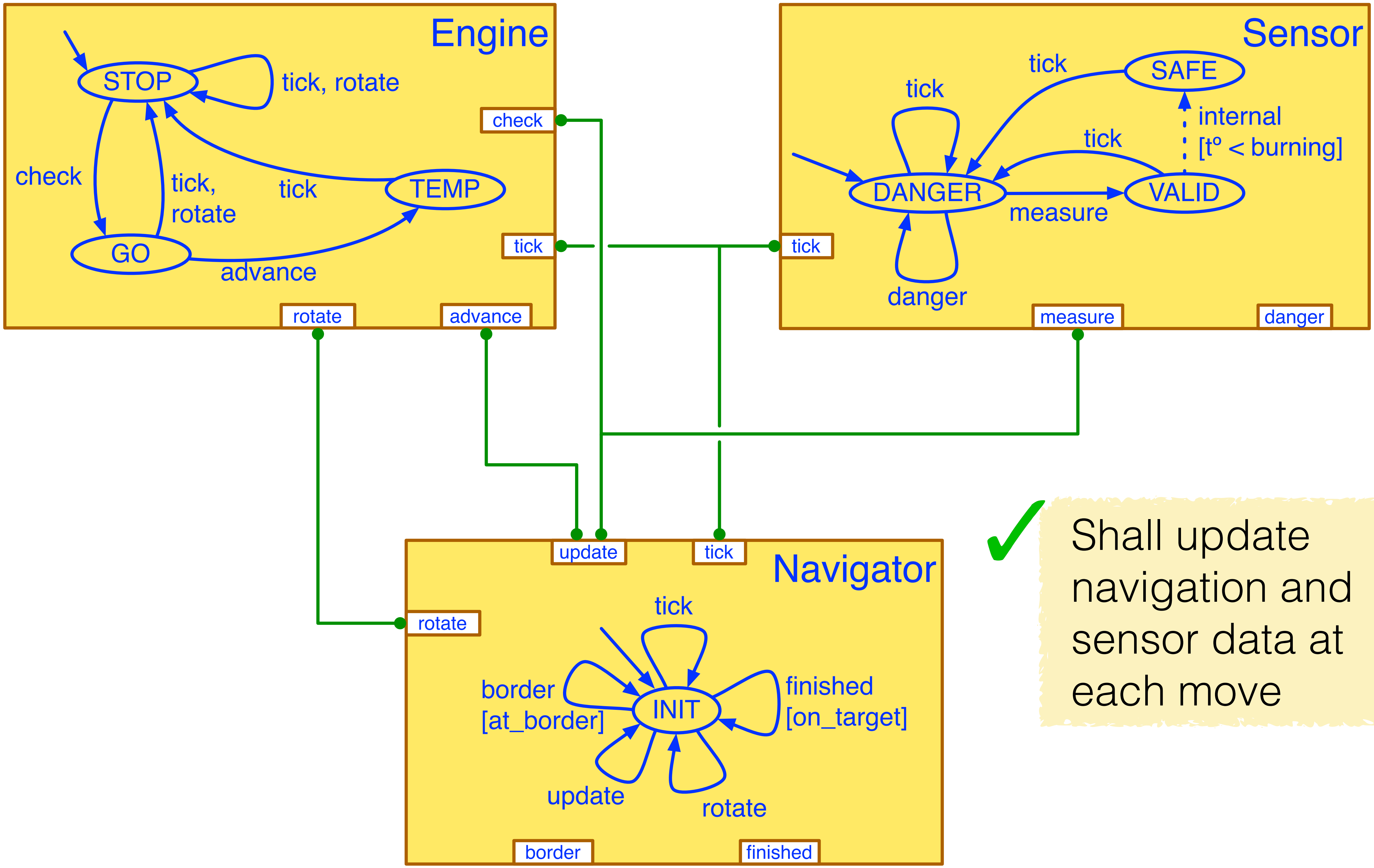


Connecting the robot



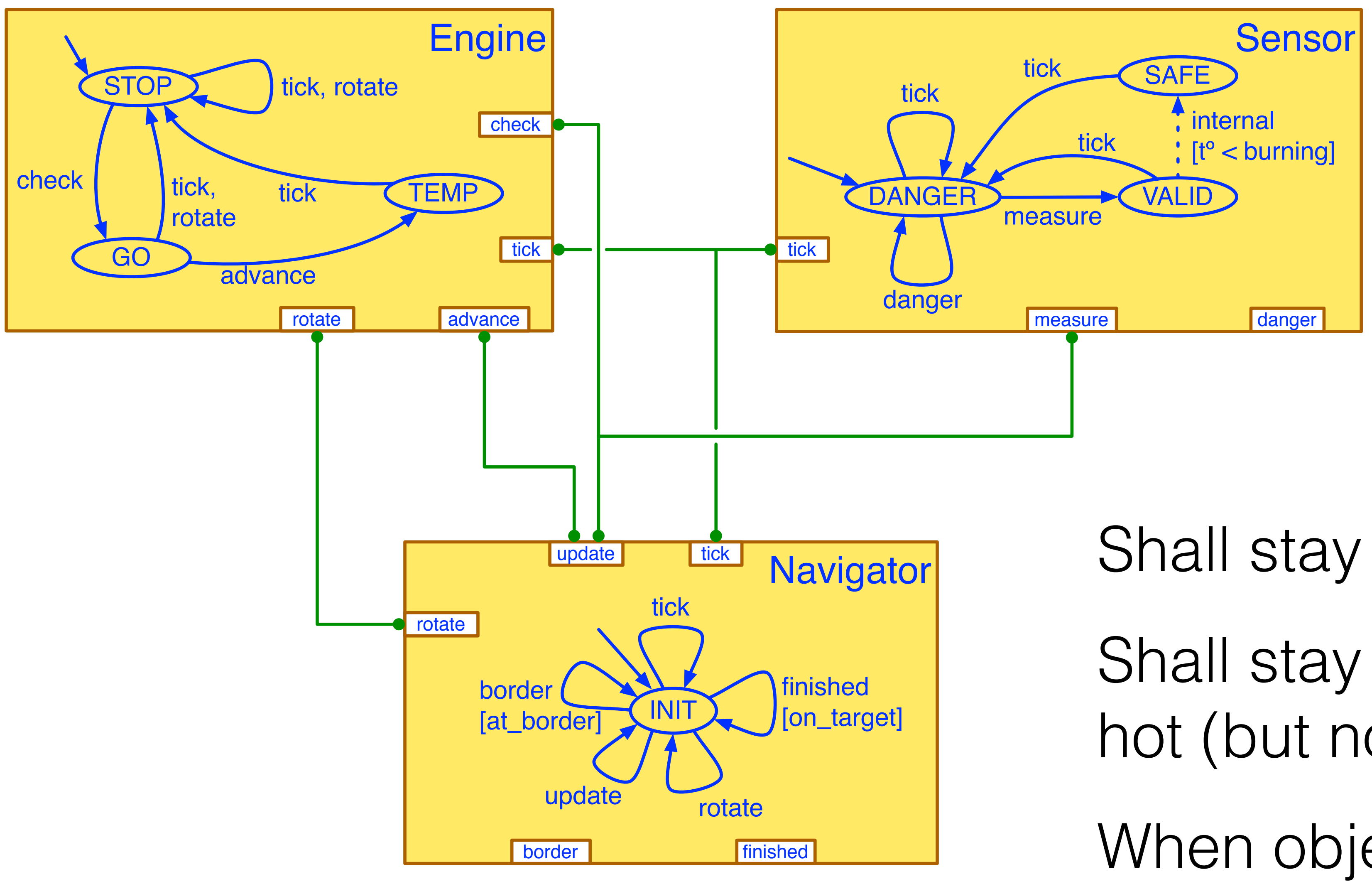
Shall update navigation and sensor data at each move

Connecting the robot



✓ Shall update navigation and sensor data at each move

Connecting the robot



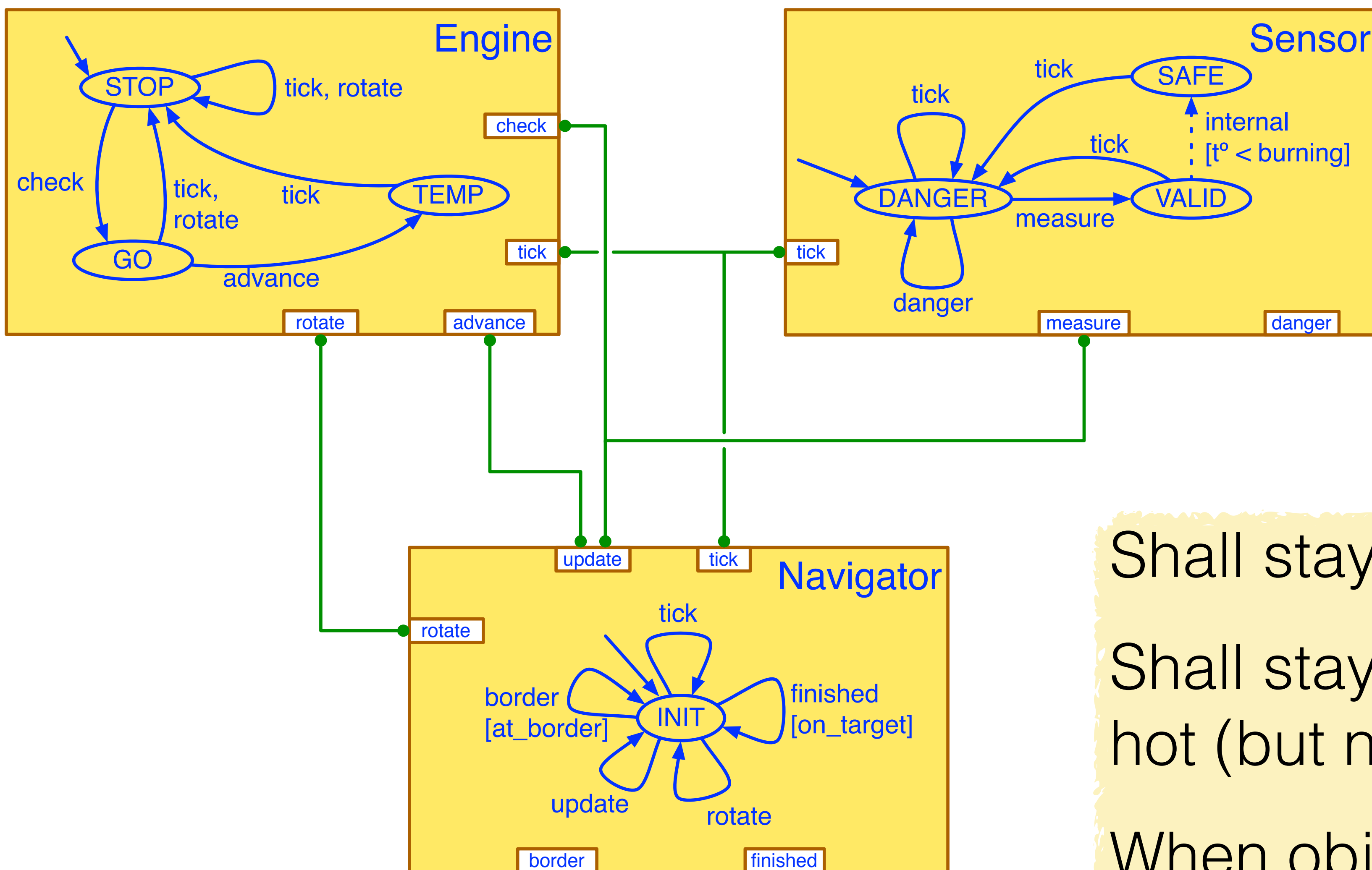
Shall stay within the region

Shall stay in the area that is safe or hot (but not burning)

When objective is found, the robot shall stop

```
priority p_rotate   c_rotate:*  < c_finished:*
priority p_advance1 c_advance:* < c_finished:*
priority p_advance2 c_advance:* < c_danger:*
priority p_advance3 c_advance:* < c_border:*
```

Connecting the robot



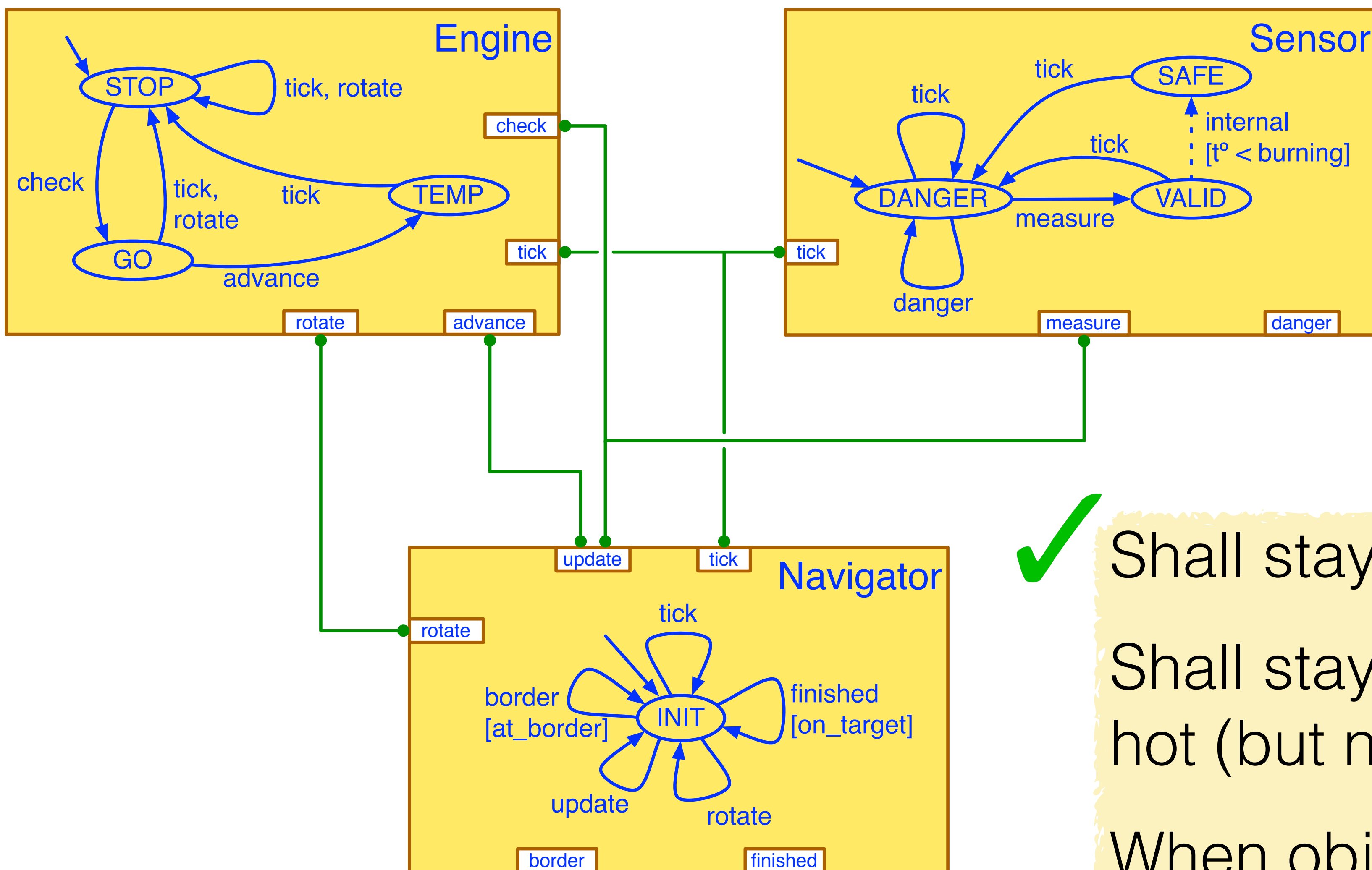
Shall stay within the region

Shall stay in the area that is safe or hot (but not burning)

When objective is found, the robot shall stop

```
priority p_rotate   c_rotate:*  < c_finished:*
priority p_advance1 c_advance:* < c_finished:*
priority p_advance2 c_advance:* < c_danger:*
priority p_advance3 c_advance:* < c_border:*
```

Connecting the robot



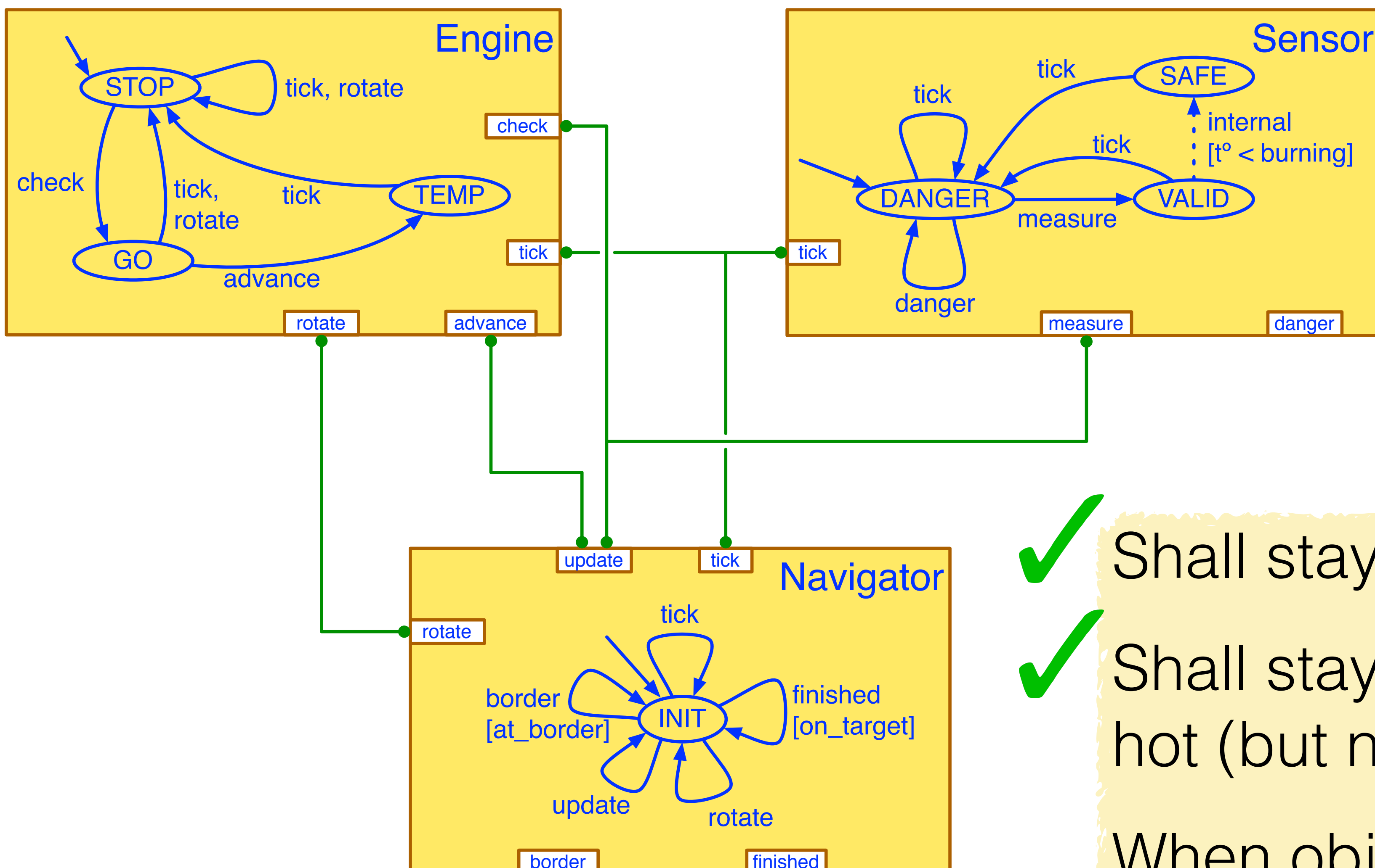
✓ Shall stay within the region

Shall stay in the area that is safe or hot (but not burning)

When objective is found, the robot shall stop

```
priority p_rotate   c_rotate:*  < c_finished:*
priority p_advance1 c_advance:* < c_finished:*
priority p_advance2 c_advance:* < c_danger:*
priority p_advance3 c_advance:* < c_border:*
```

Connecting the robot



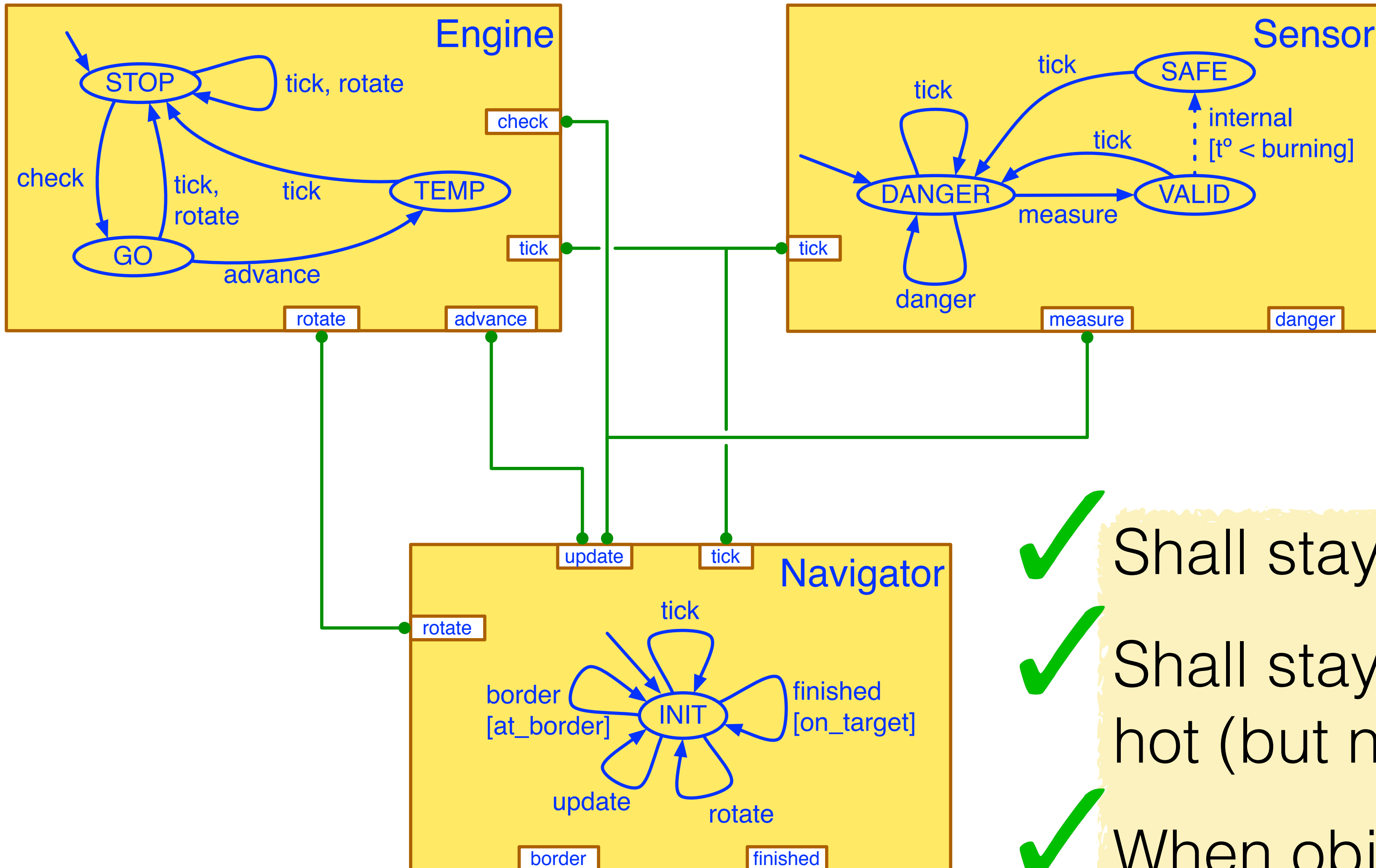
✓ Shall stay within the region

✓ Shall stay in the area that is safe or hot (but not burning)

When objective is found, the robot shall stop

```
priority p_rotate   c_rotate:*  < c_finished:*
priority p_advance1 c_advance:* < c_finished:*
priority p_advance2 c_advance:* < c_danger:*
priority p_advance3 c_advance:* < c_border:*
```

Connecting the robot



✓ Shall stay within the region

✓ Shall stay in the area that is safe or hot (but not burning)

✓ When objective is found, the robot shall stop

```
priority p_rotate   c_rotate:*  < c_finished:*
priority p_advance1 c_advance:* < c_finished:*
priority p_advance2 c_advance:* < c_danger:*
priority p_advance3 c_advance:* < c_border:*
```

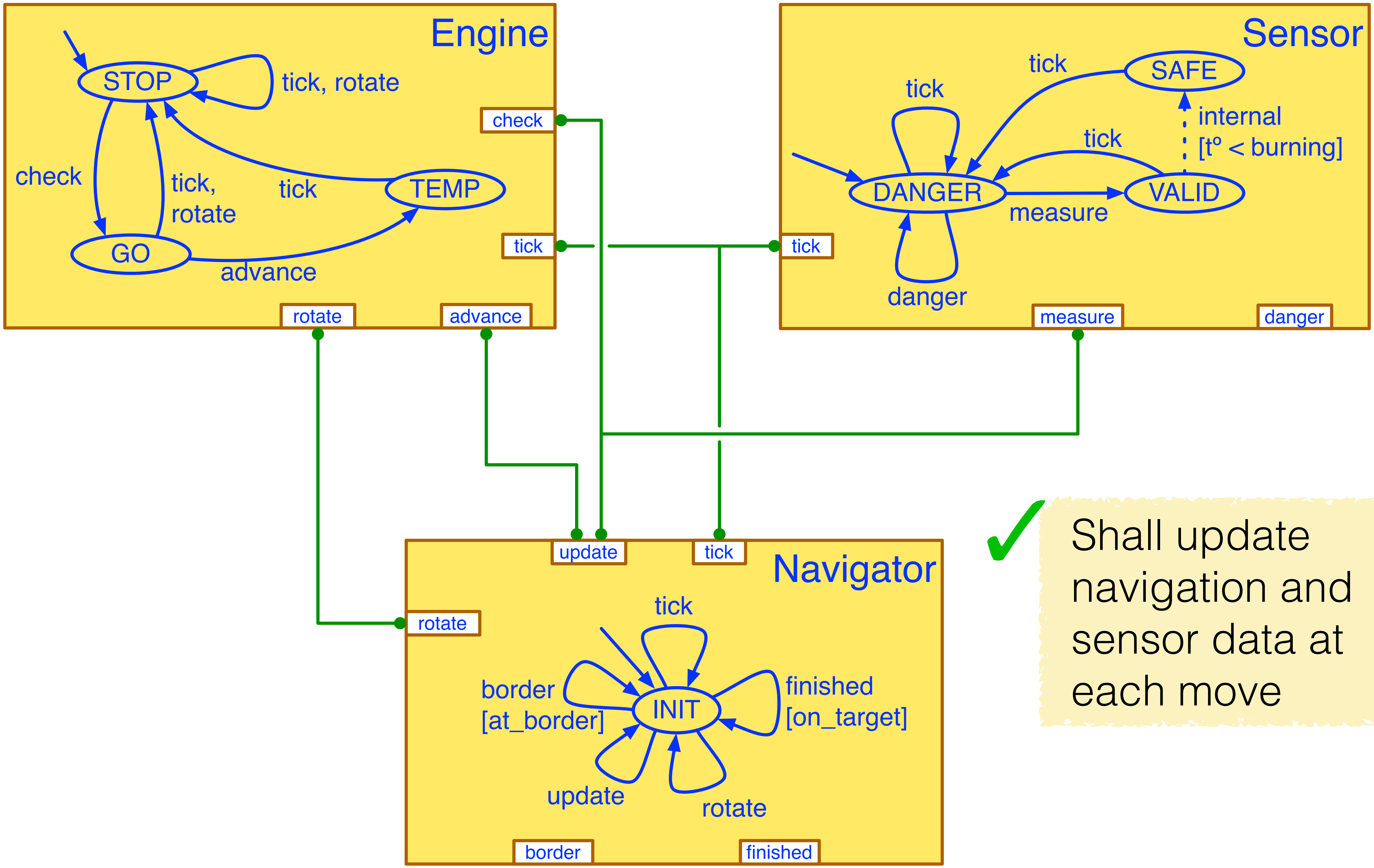
The final step

Remove the model of the environment

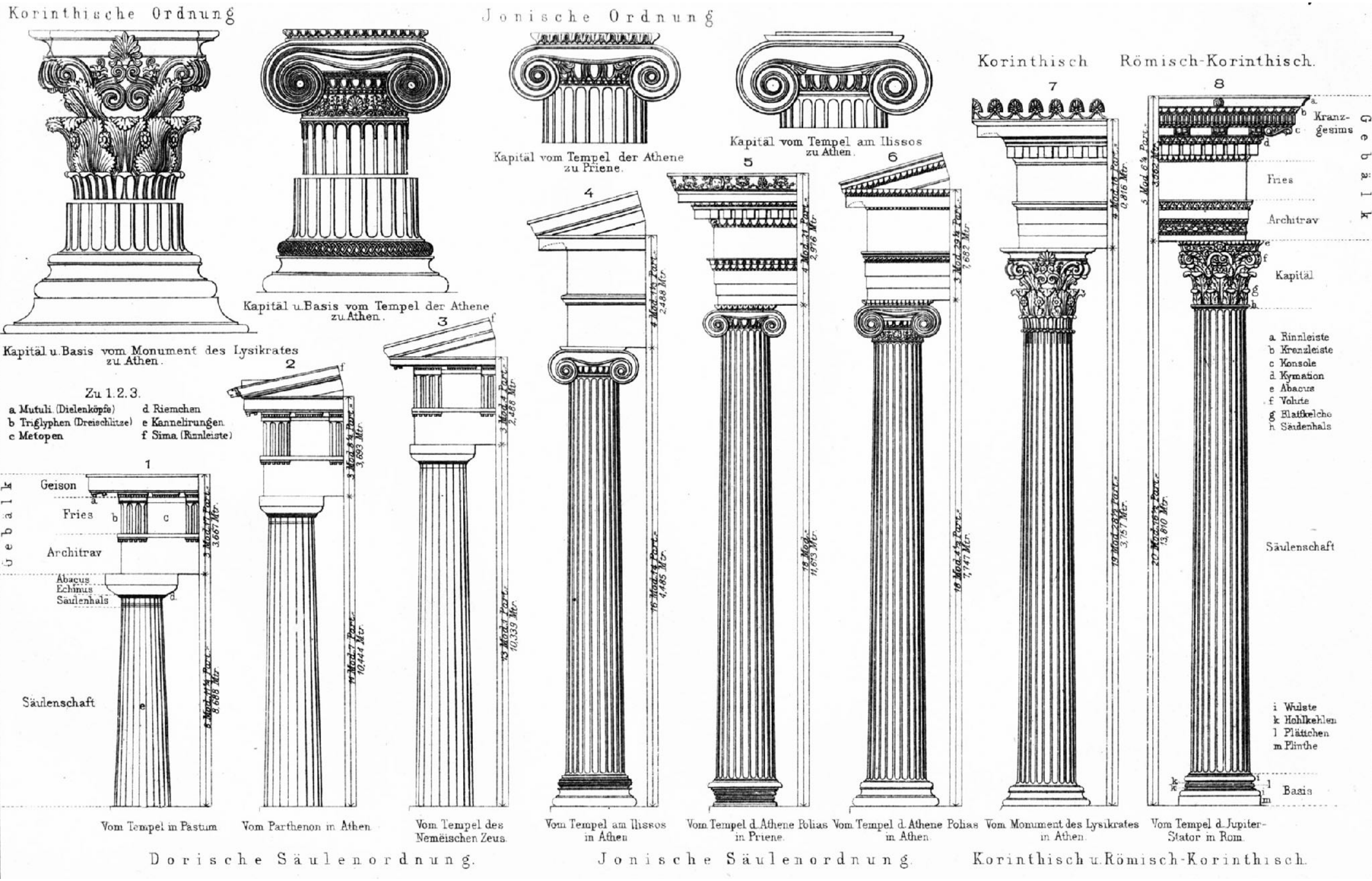
Replace “interface” elements with corresponding primitives

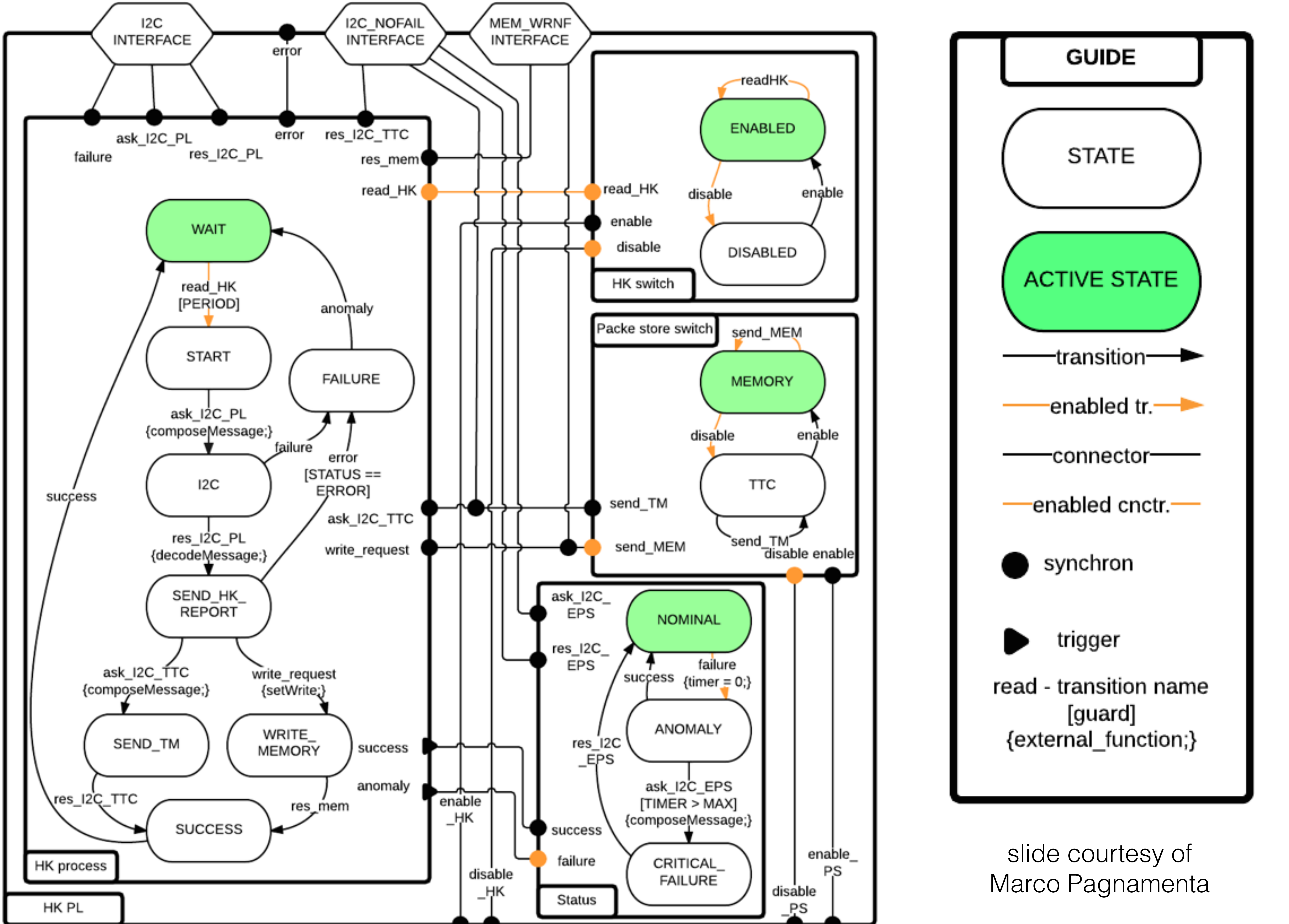
Generate executable code from the remaining model

Connecting the robot

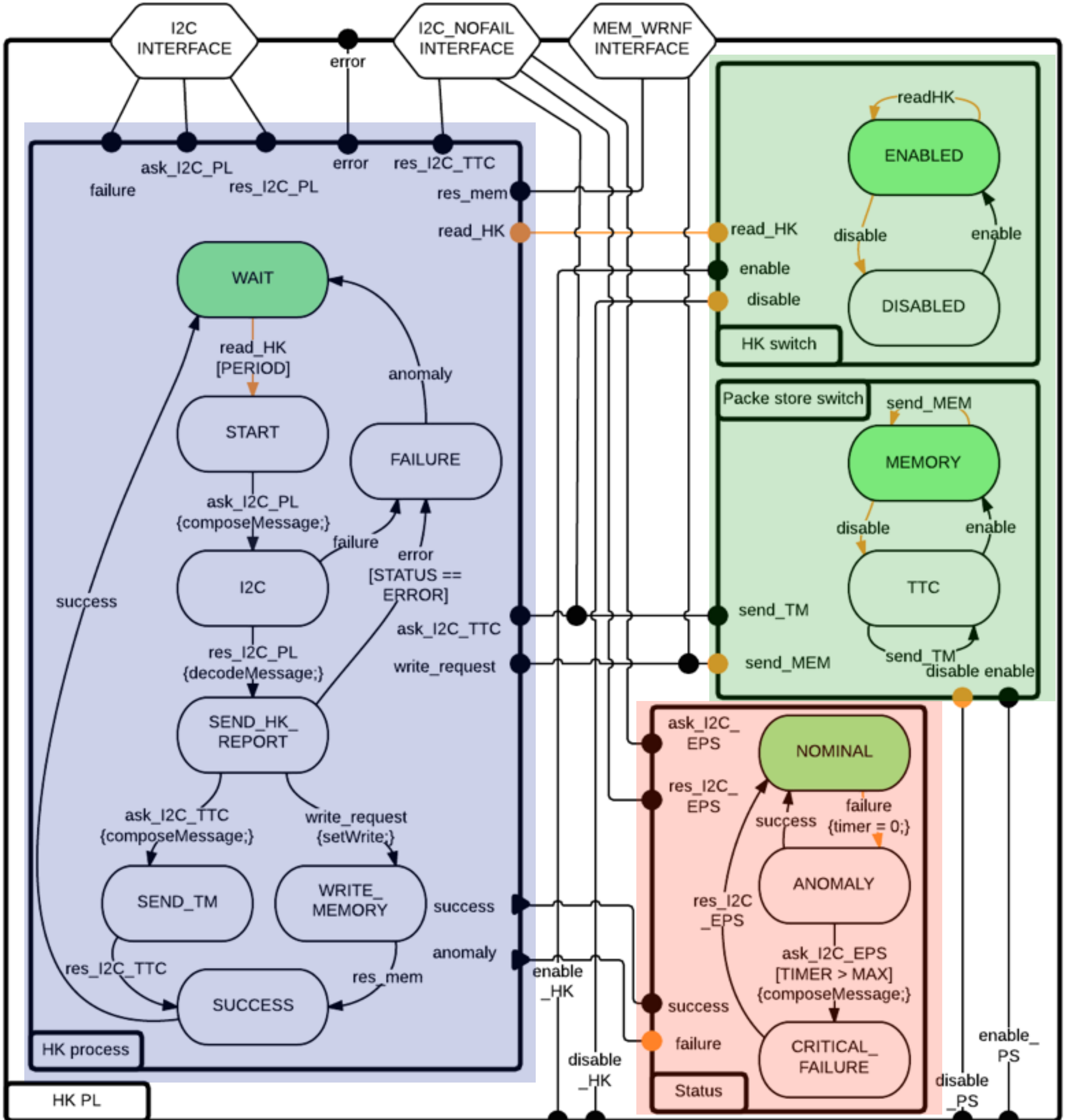


✓ Shall update navigation and sensor data at each move


Korinthische Ordnung
Jonische Ordnung
Kapitäl u. Basis vom Monument des Lysikrates zu Athen.
Kapitäl u. Basis vom Tempel der Athene zu Athen.
Kapitäl vom Tempel der Athene zu Priene.
Kapitäl vom Tempel am Ilissos zu Athen.
Korinthisch
Römisch-Korinthisch.
Zu 1.2.3.
a Mutuli (Dielenköpfe)
b Triglyphen (Dreischlitze)
c Metopen
d Riemchen
e Kannelirungen
f Sima (Rinnleiste)
1
2
3
4
5
6
7
8
Gebälk
Geison
Fries
Architrav
Abacus
Echinus
Säulenhals
Säulenschaft
Kranz-gesims
Fries
Architrav
Kapitäl
Säulenschaft
Basis
Gebälk
a Rinnleiste
b Kranzleiste
c Konsole
d Kymation
e Abacus
f Voluten
g Blattkelche
h Säulenhals
i Wulste
k Hohlkehlen
l Plättchen
m Plinthe
Vom Tempel in Pästum
Vom Parthenon in Athen
Vom Tempel des Nemëischen Zeus.
Vom Tempel am Ilissos in Athen
Vom Tempel d. Athene Polias in Priene.
Vom Tempel d. Athene Polias in Athen.
Vom Monument des Lysikrates in Athen.
Vom Tempel d. Jupiter-Stator in Rom.
Dorische Säulenordnung.
Jonische Säulenordnung.
Korinthisch u. Römisch-Korinthisch.



slide courtesy of Marco Pagnamenta



GUIDE

- STATE
- ACTIVE STATE
- transition
- enabled tr.
- connector
- enabled cnctr.
- synchron
- trigger

read - transition name
[guard]
{external_function;}

slide courtesy of Marco Pagnamenta

Theory of architectures

Design patterns for BIP

How to model?

How to combine?

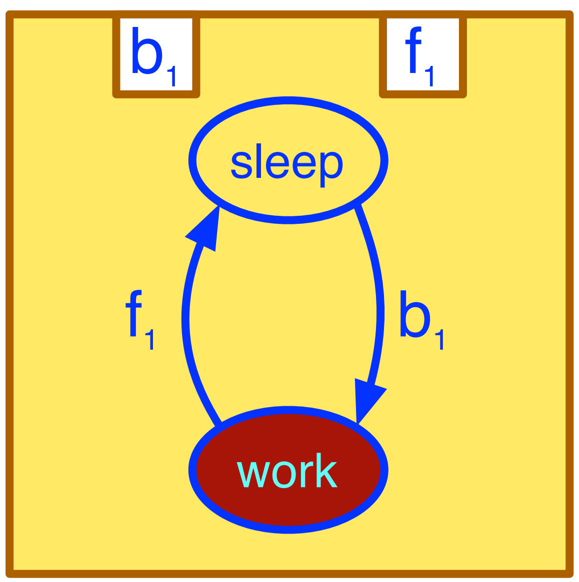
How to specify?

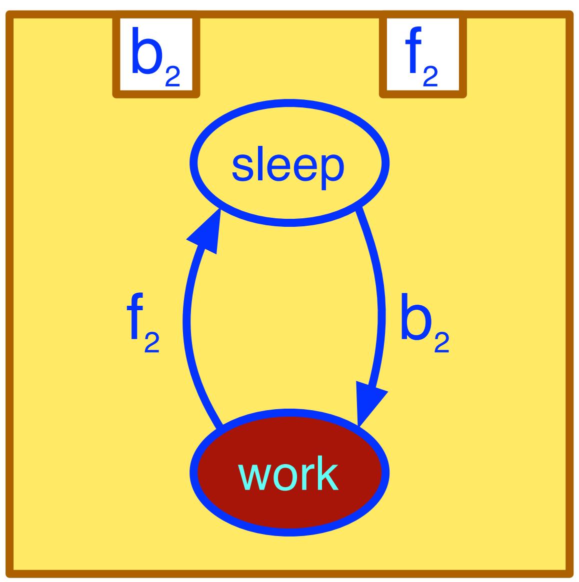
Architectures enforce characteristic properties. The crucial question is whether these are preserved by composition?

[Attie et al, SEFM '14]

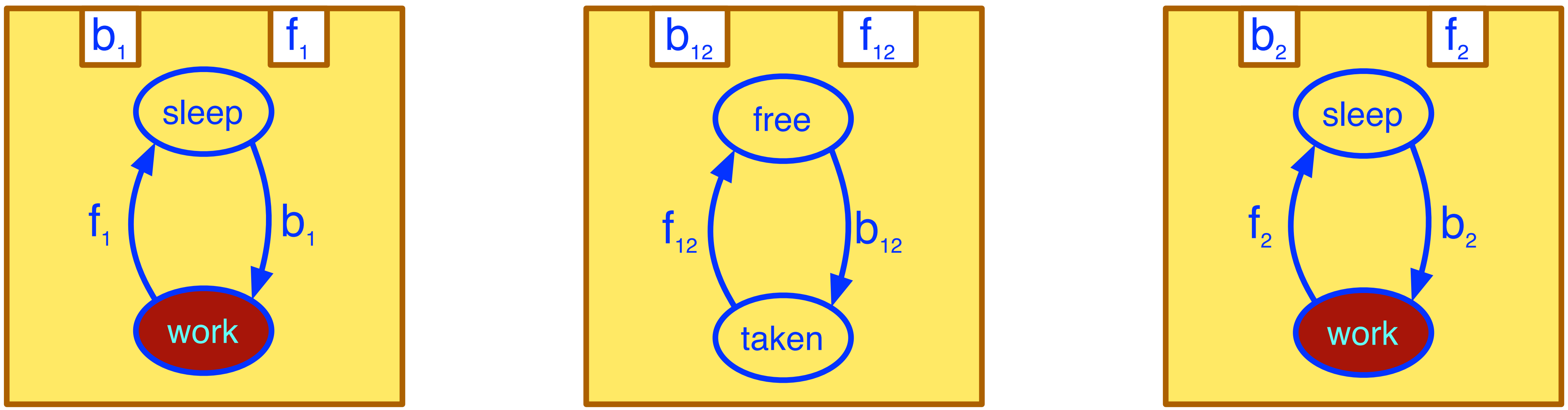
How to model?

Example: Lock

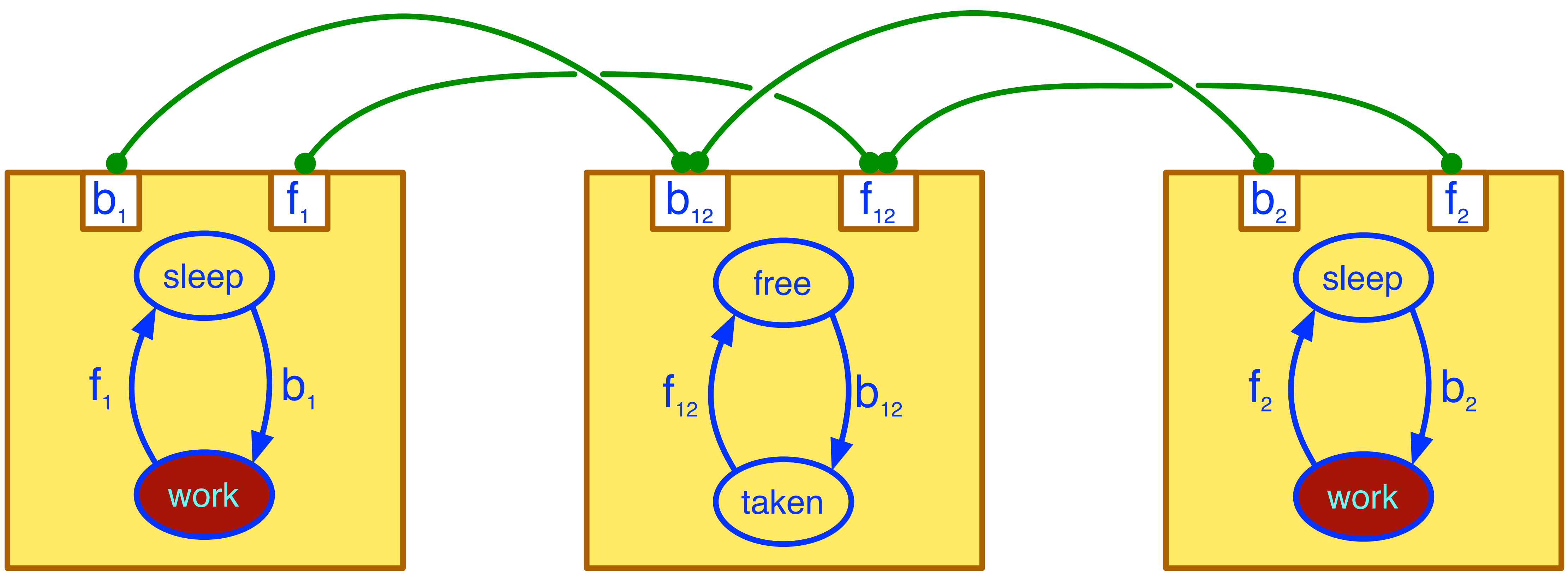




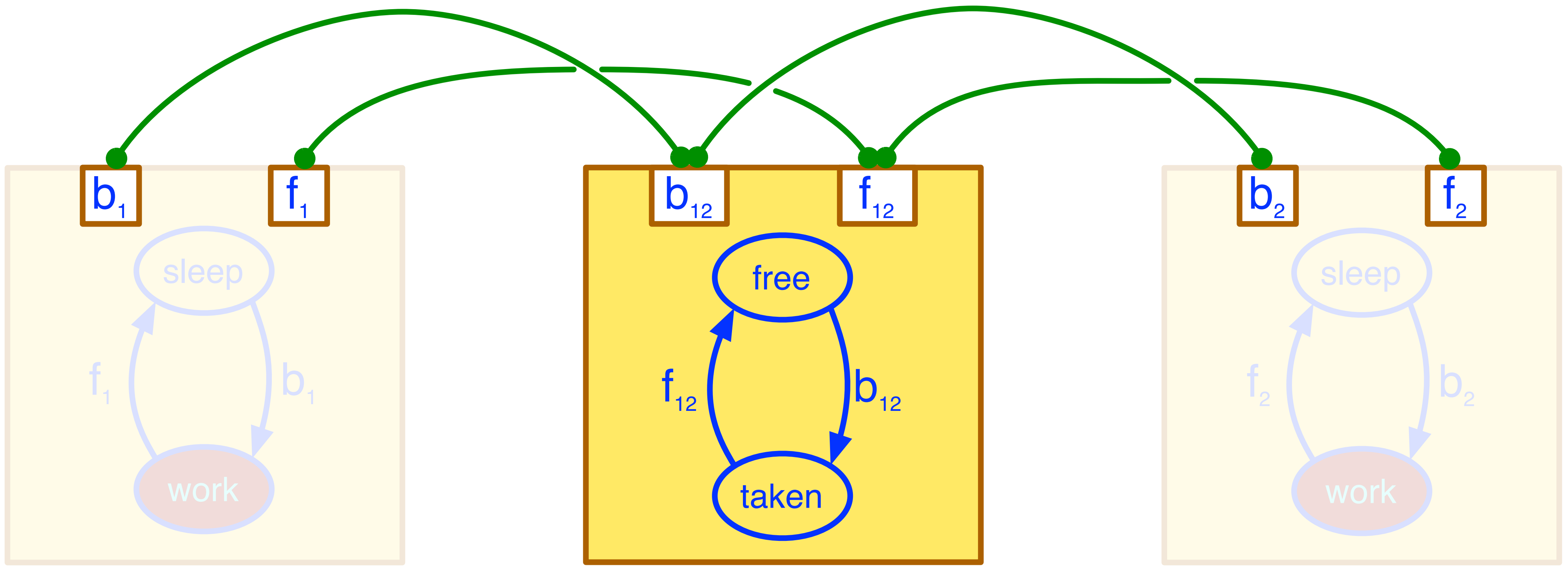
Example: Lock



Example: Lock



Example: Lock



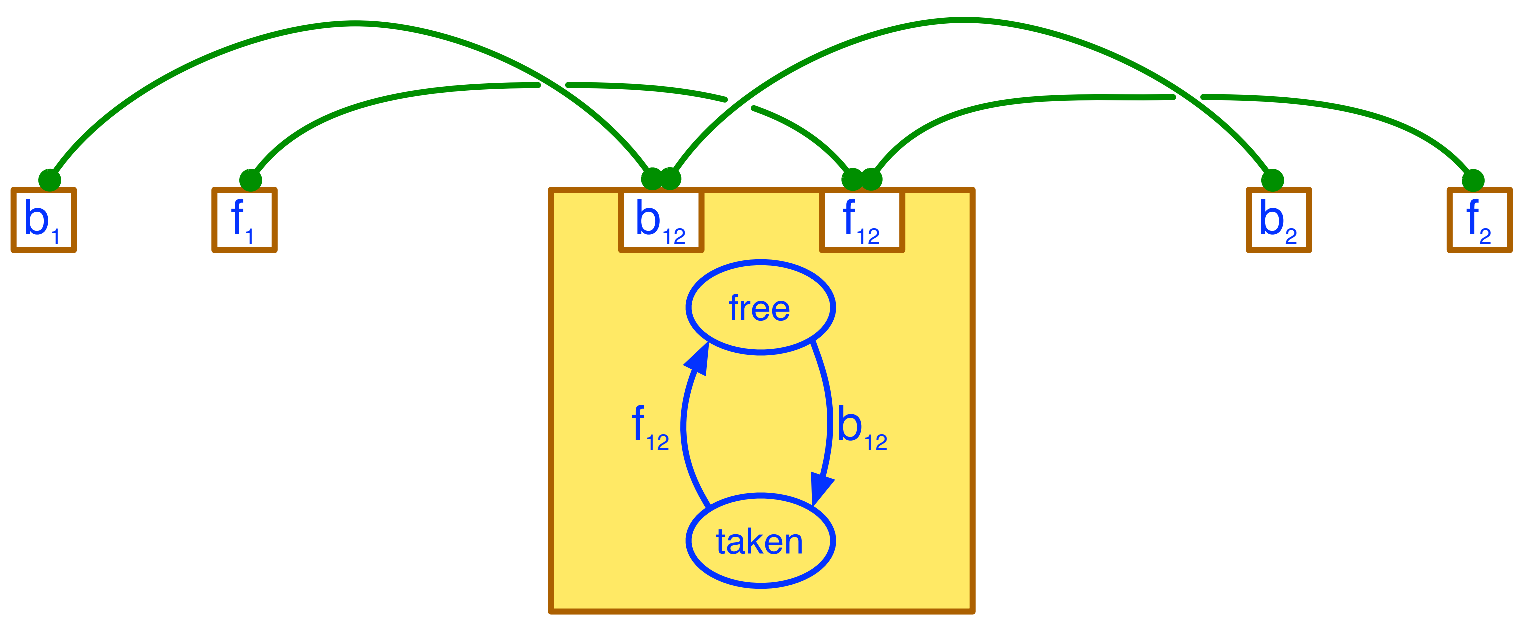
An architecture is...

$$A = (\mathcal{C}, P_A, \gamma)$$

Set of coordinating behaviours

Interface (ports)

Interaction model



…an operator…

$$A = (\mathcal{C}, P_A, \gamma)$$

…transforming

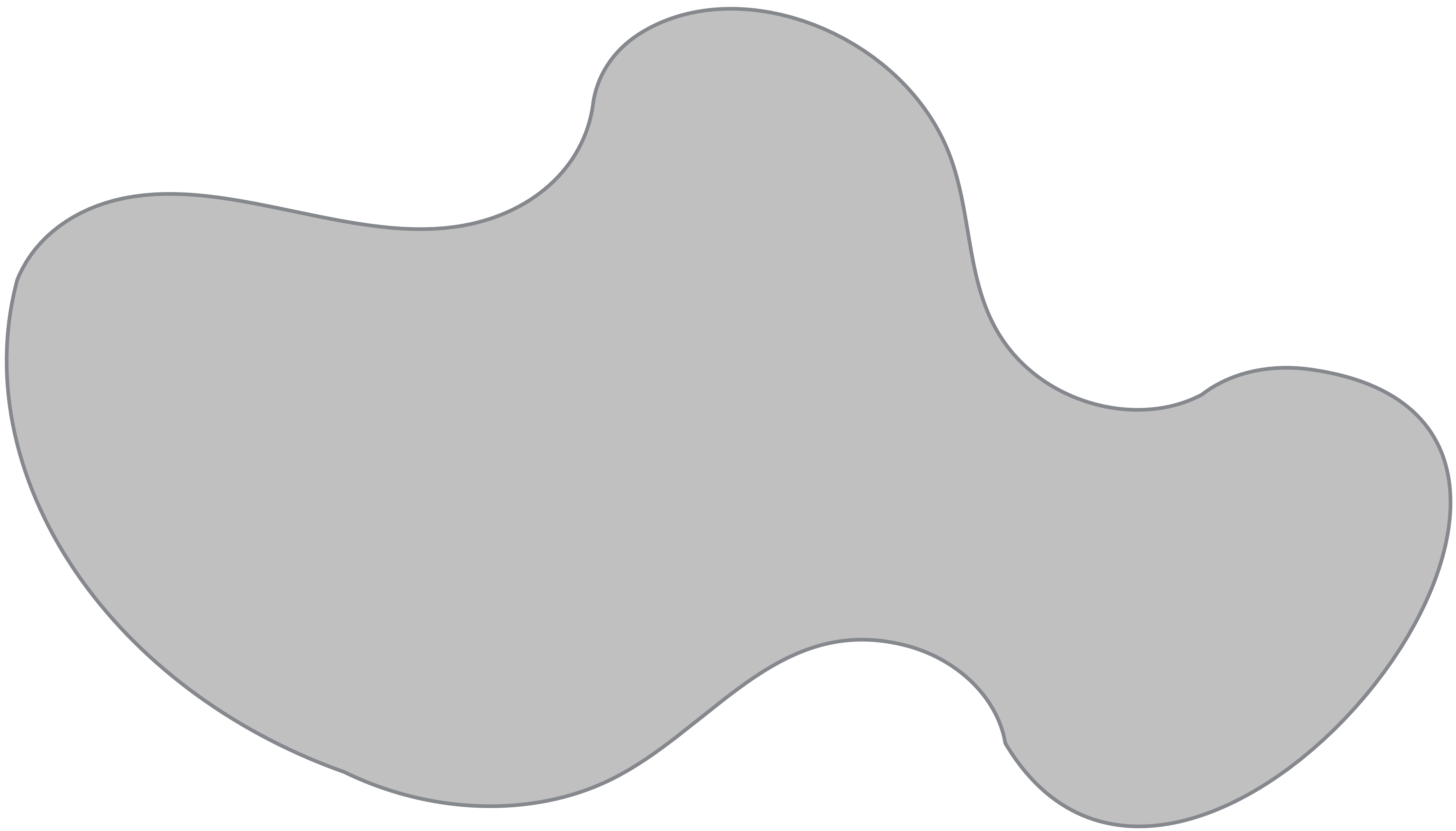
a set of components $\mathcal{B}$

into a composed BIP system $\quad A(\mathcal{B}) \stackrel{def}{=} \Big(\gamma \ltimes P\Big)(\mathcal{B} \cup \mathcal{C})$

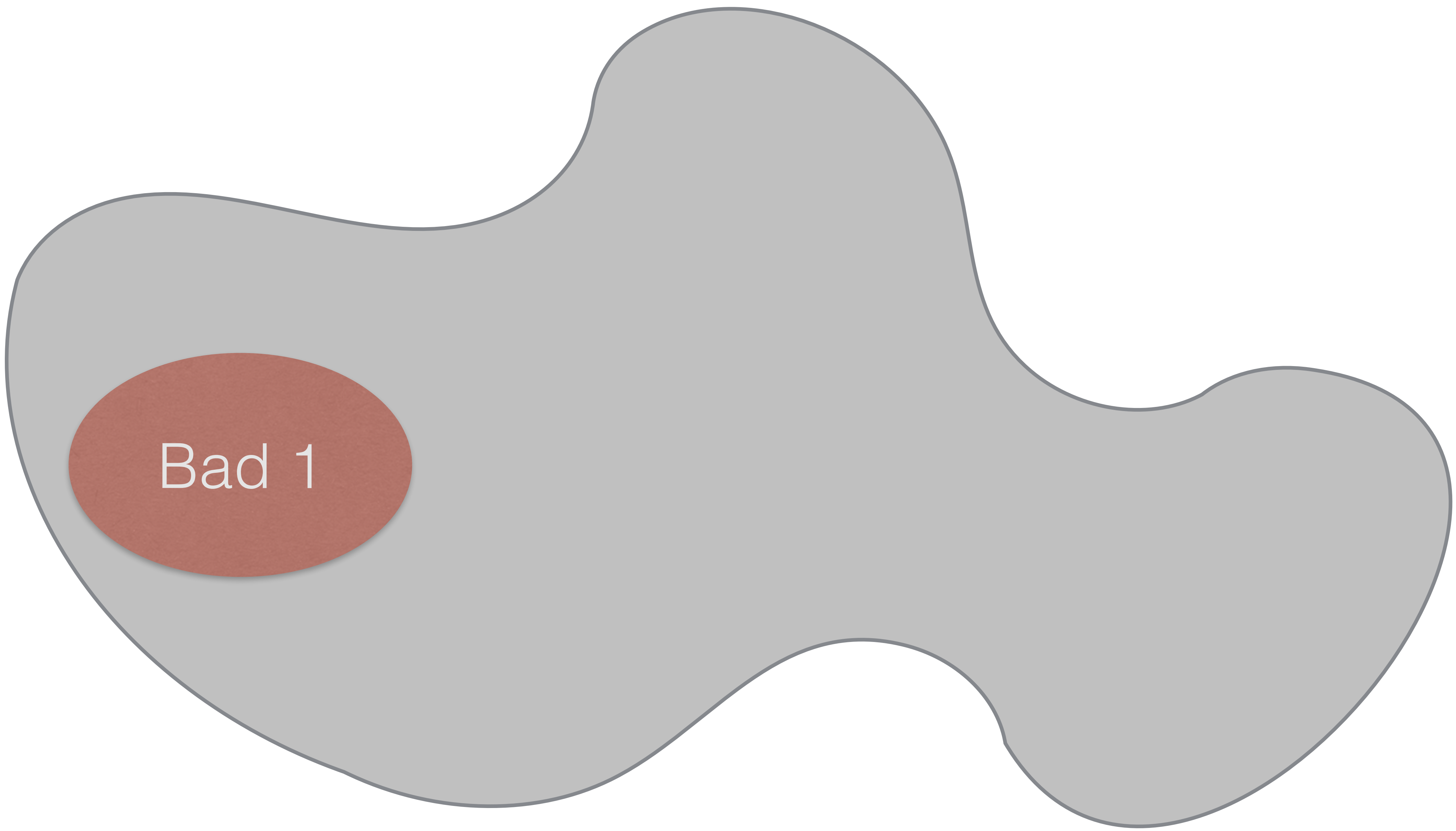
where $\quad P \stackrel{def}{=} \bigcup_{B \in \mathcal{B} \cup \mathcal{C}} P_B, \quad \gamma \ltimes P \stackrel{def}{=} \left\{a \subseteq 2^P \,\middle|\, a \cap P_A \in \gamma\right\}$

How to combine?

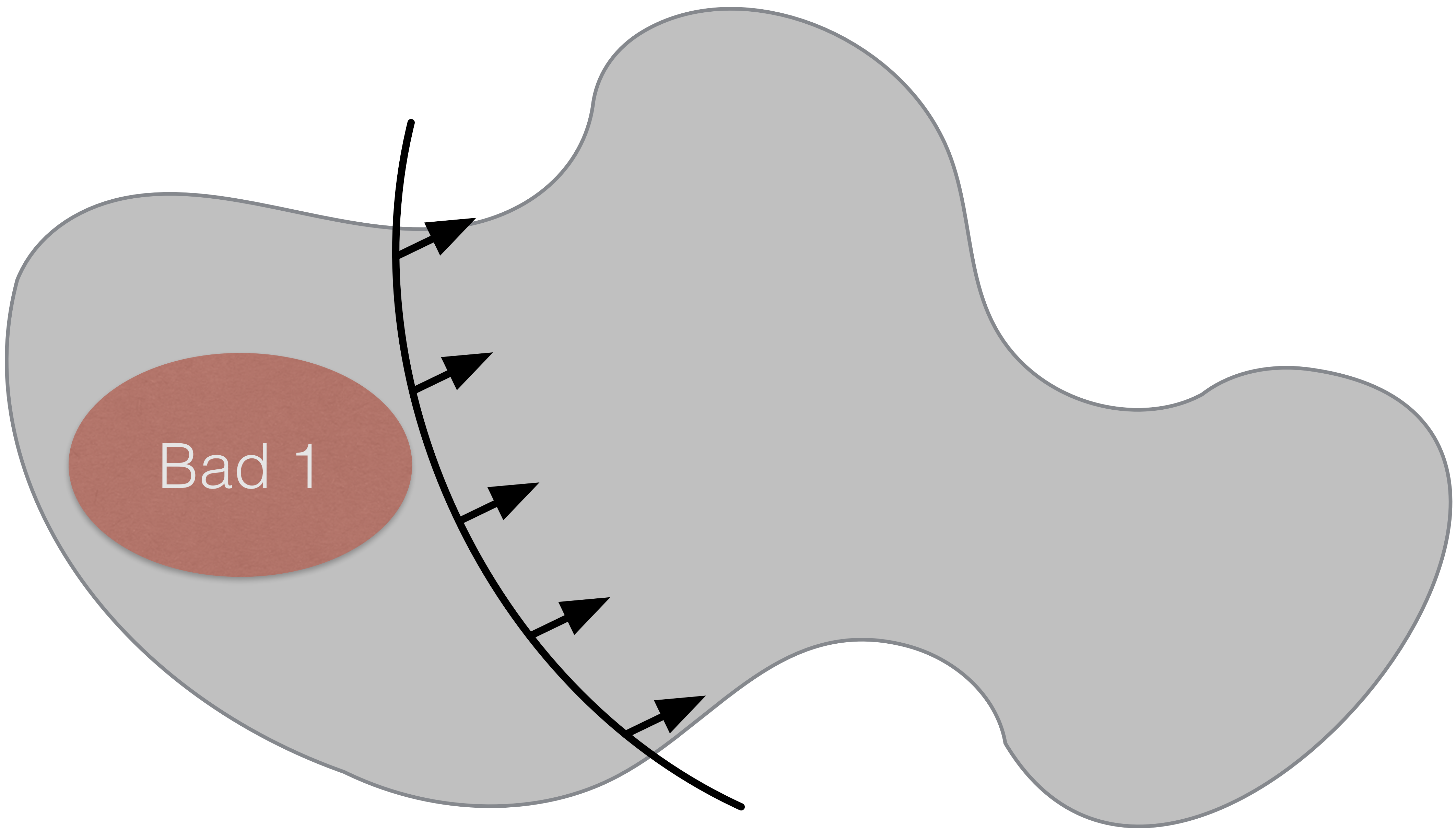
Constraints intuition



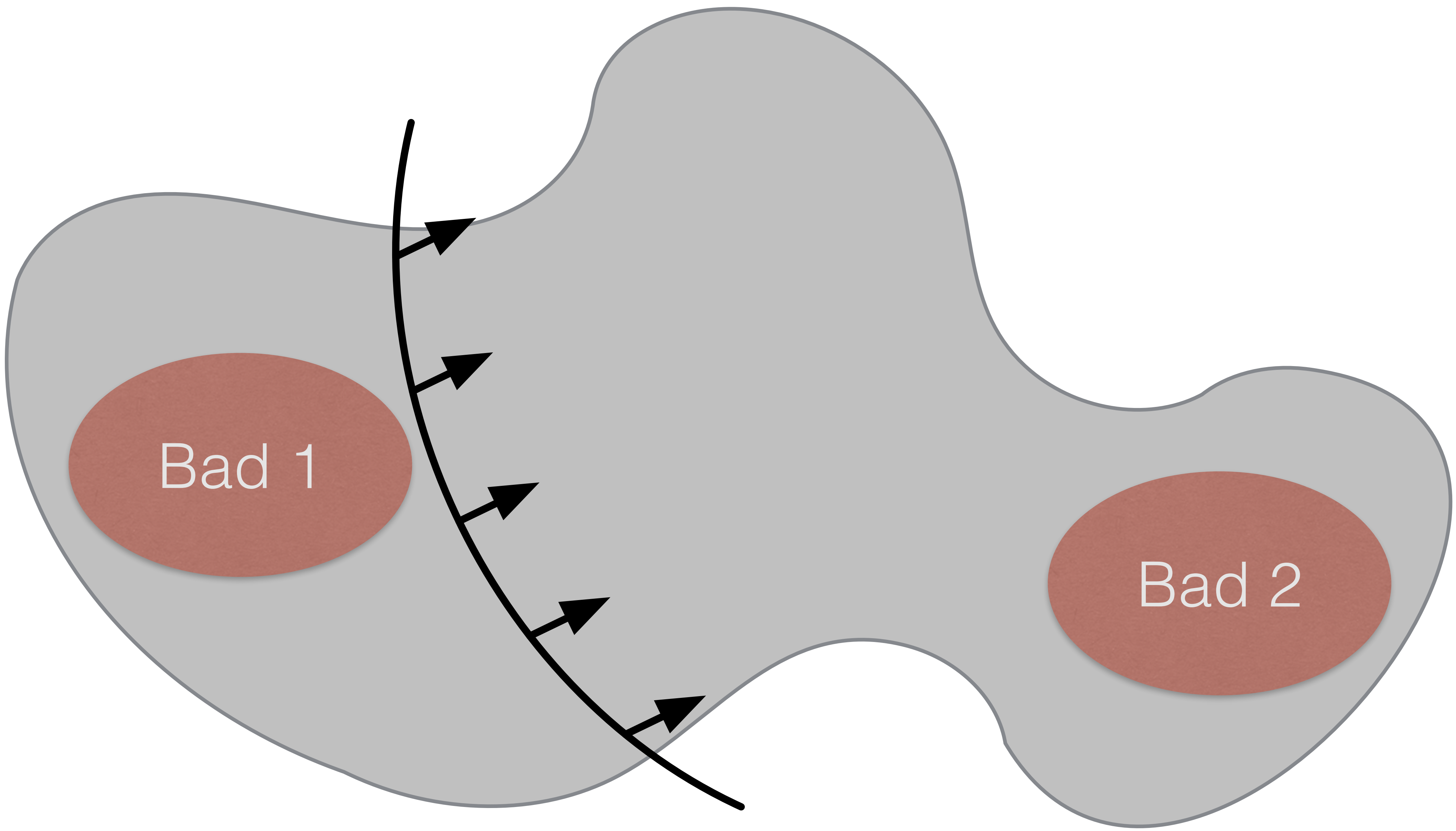
Constraints intuition



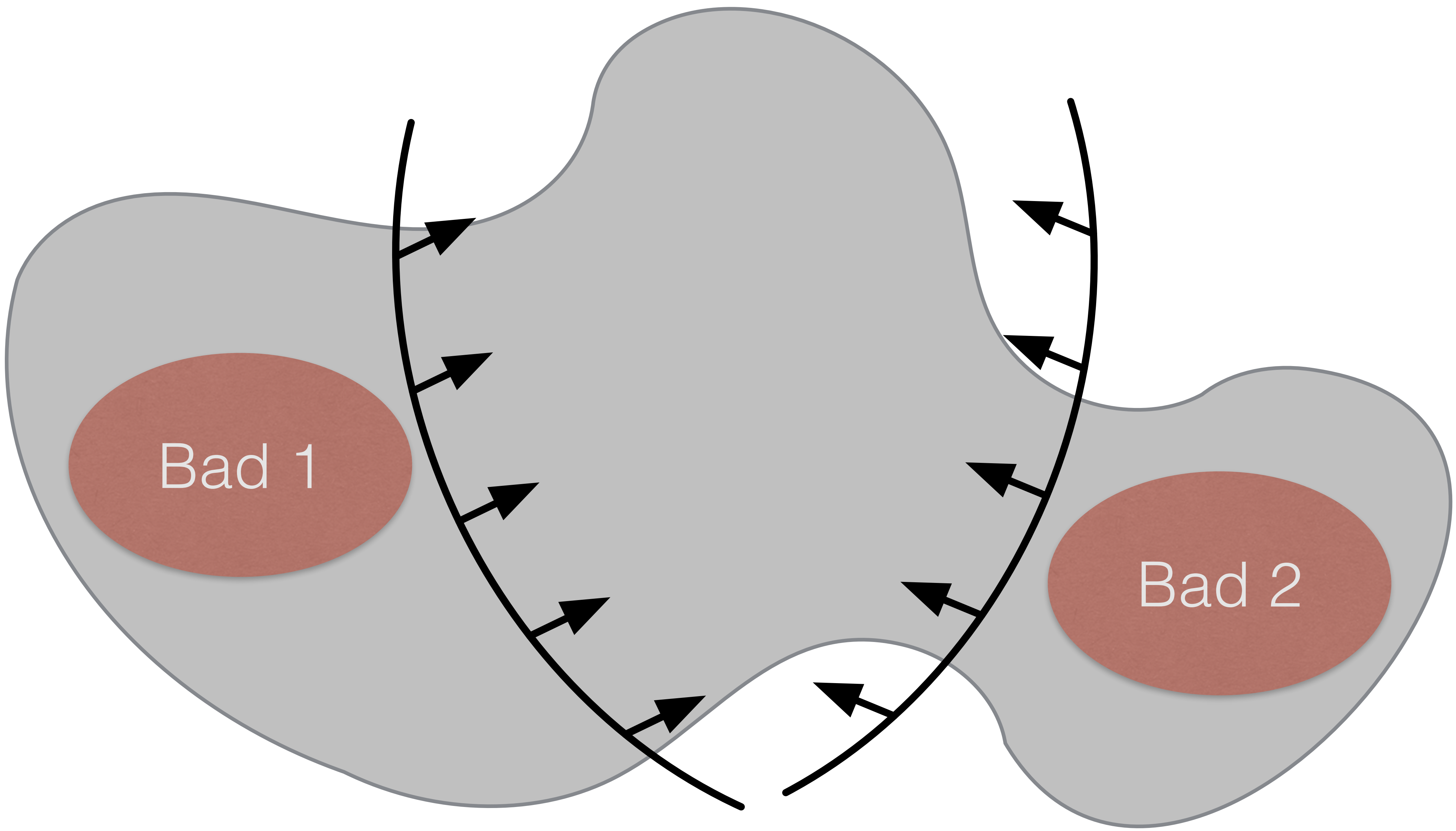
Constraints intuition



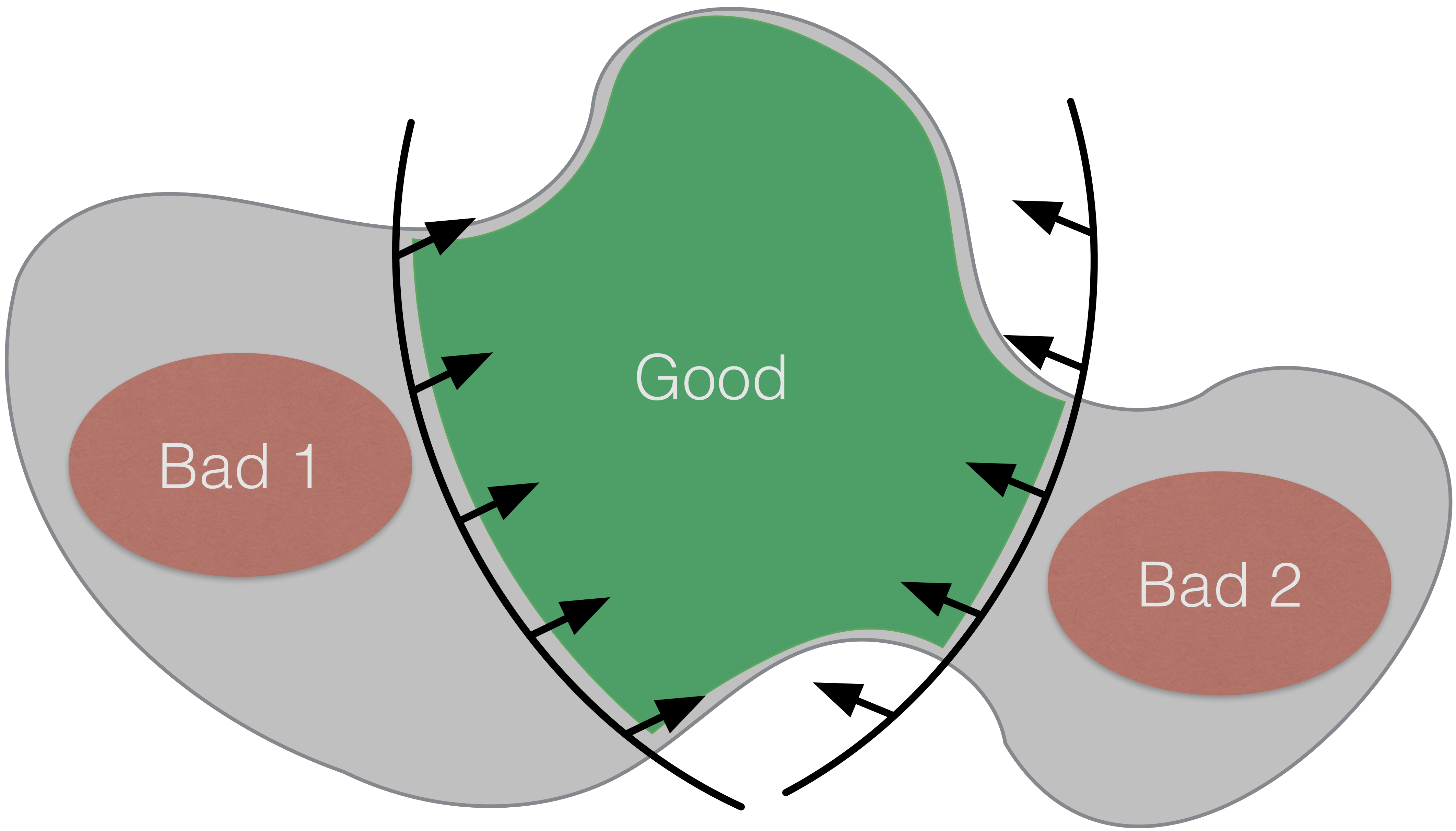
Constraints intuition



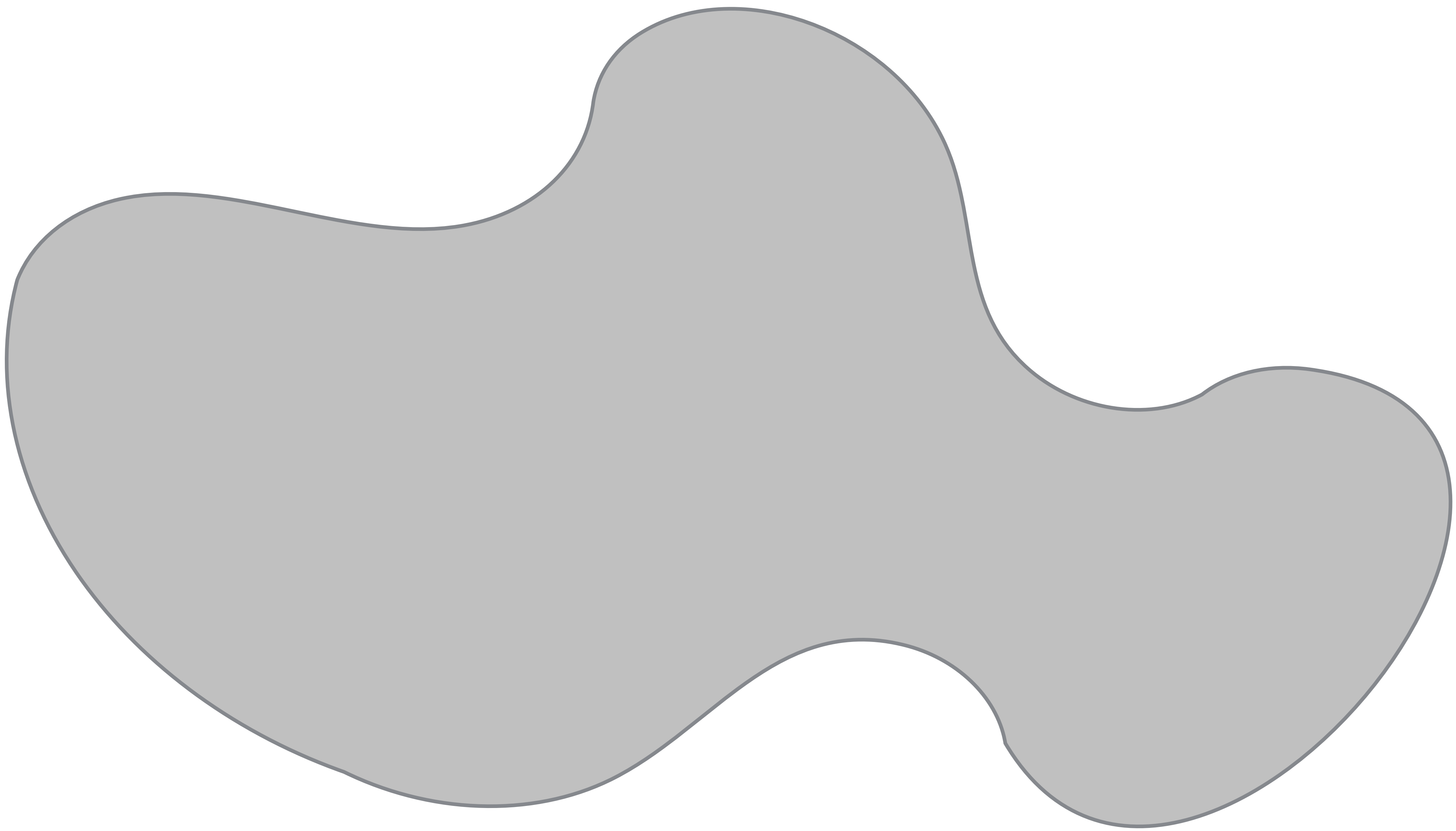
Constraints intuition



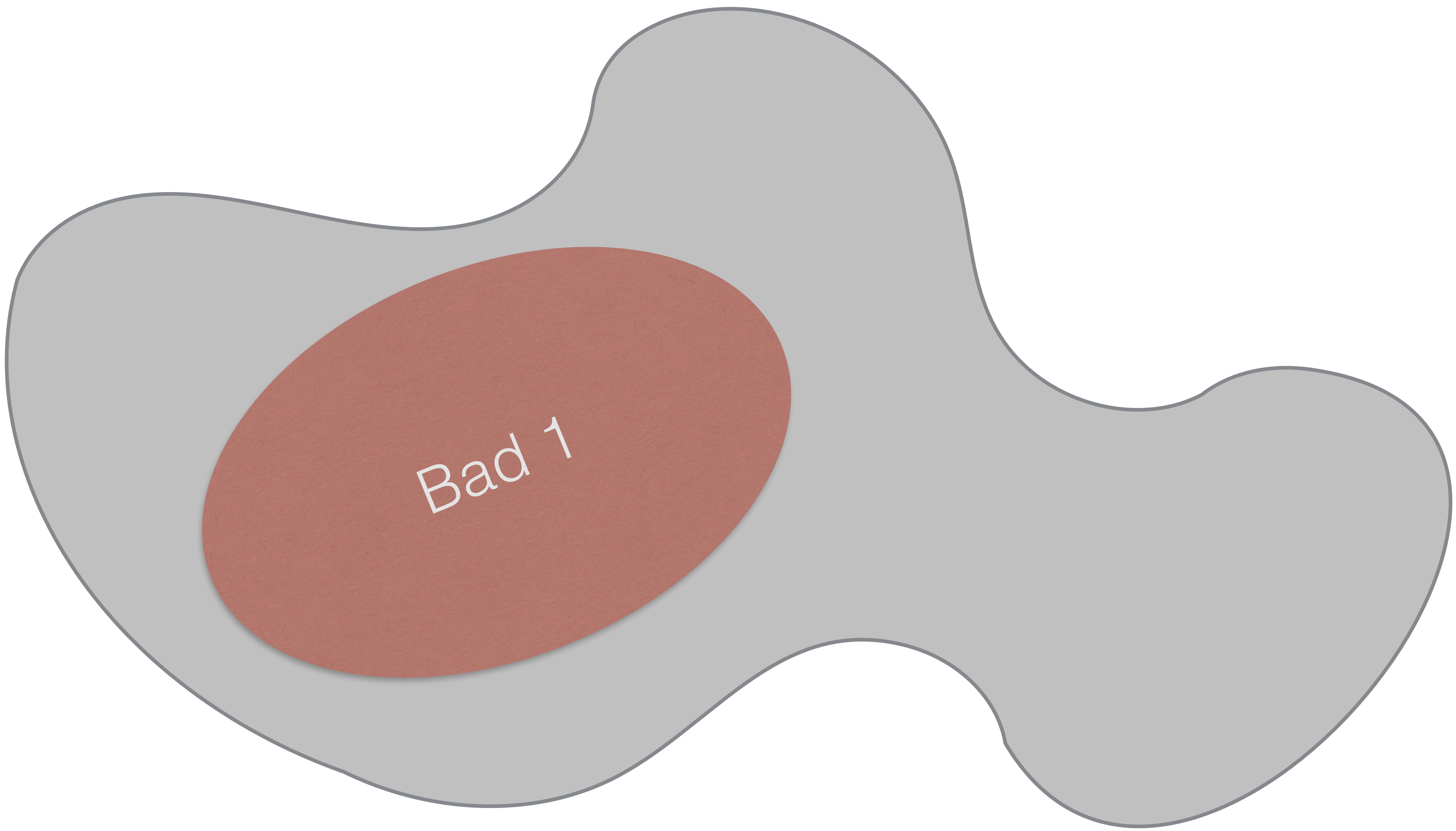
Constraints intuition



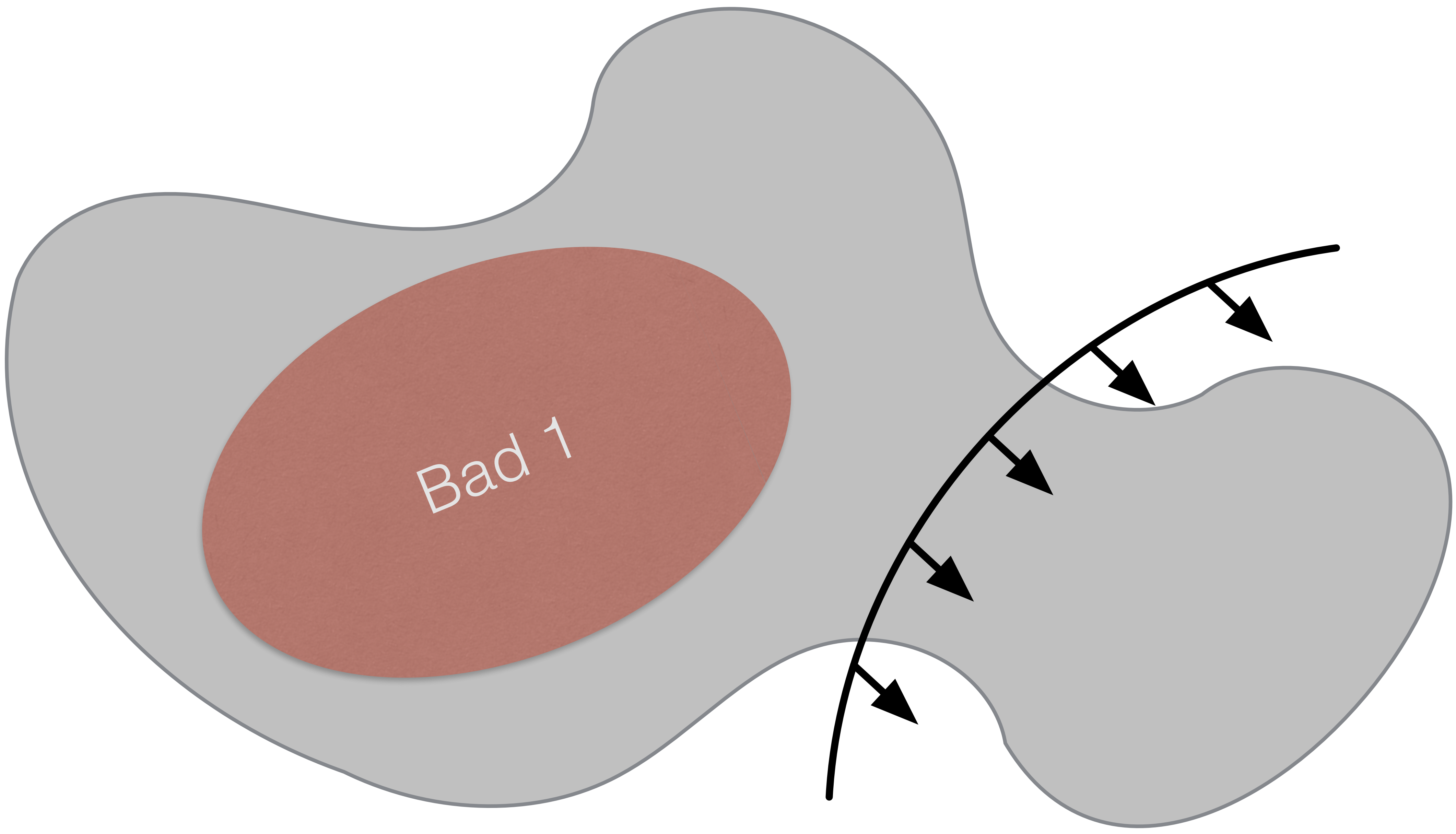
Limits of white magic



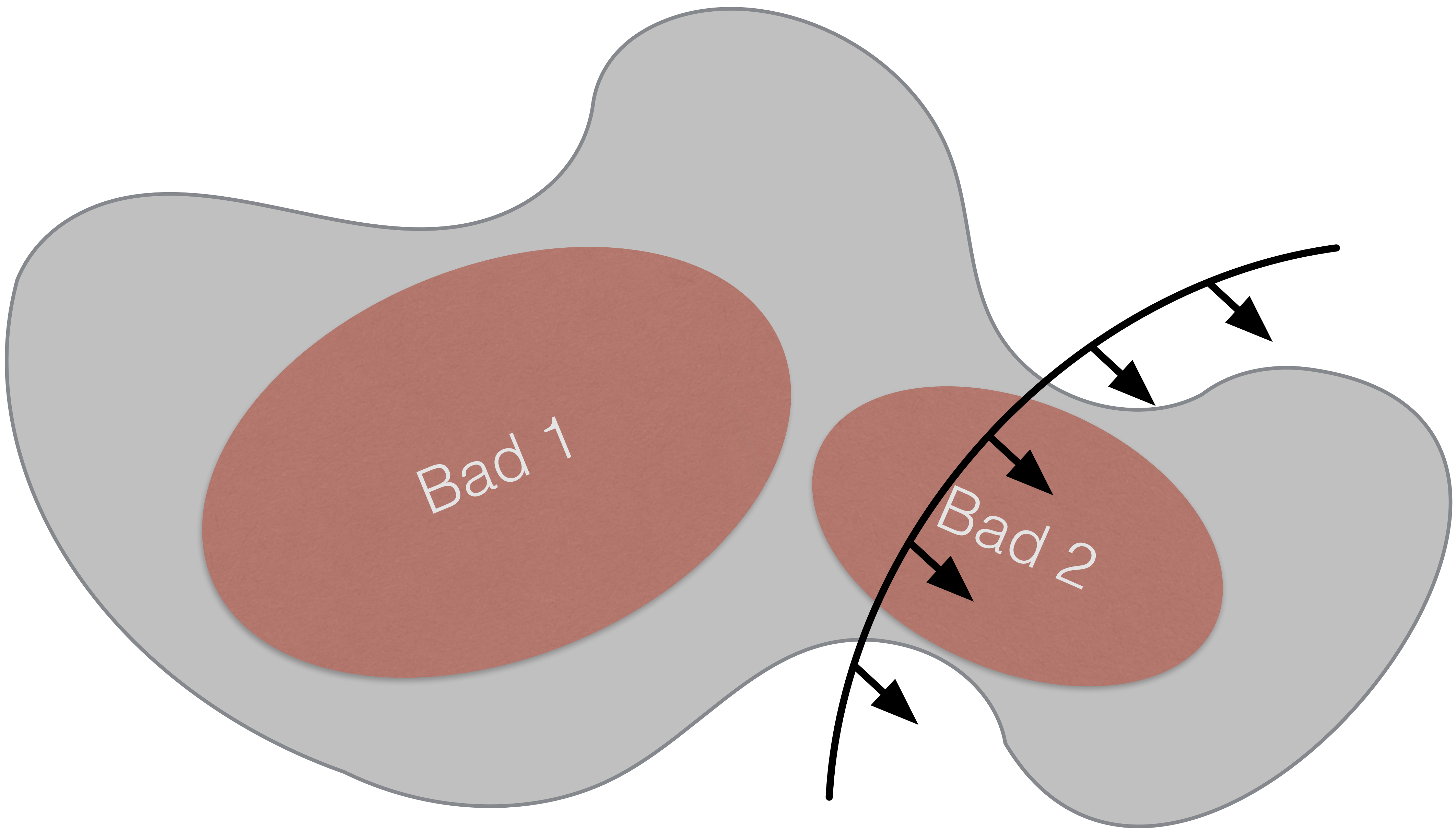
Limits of white magic



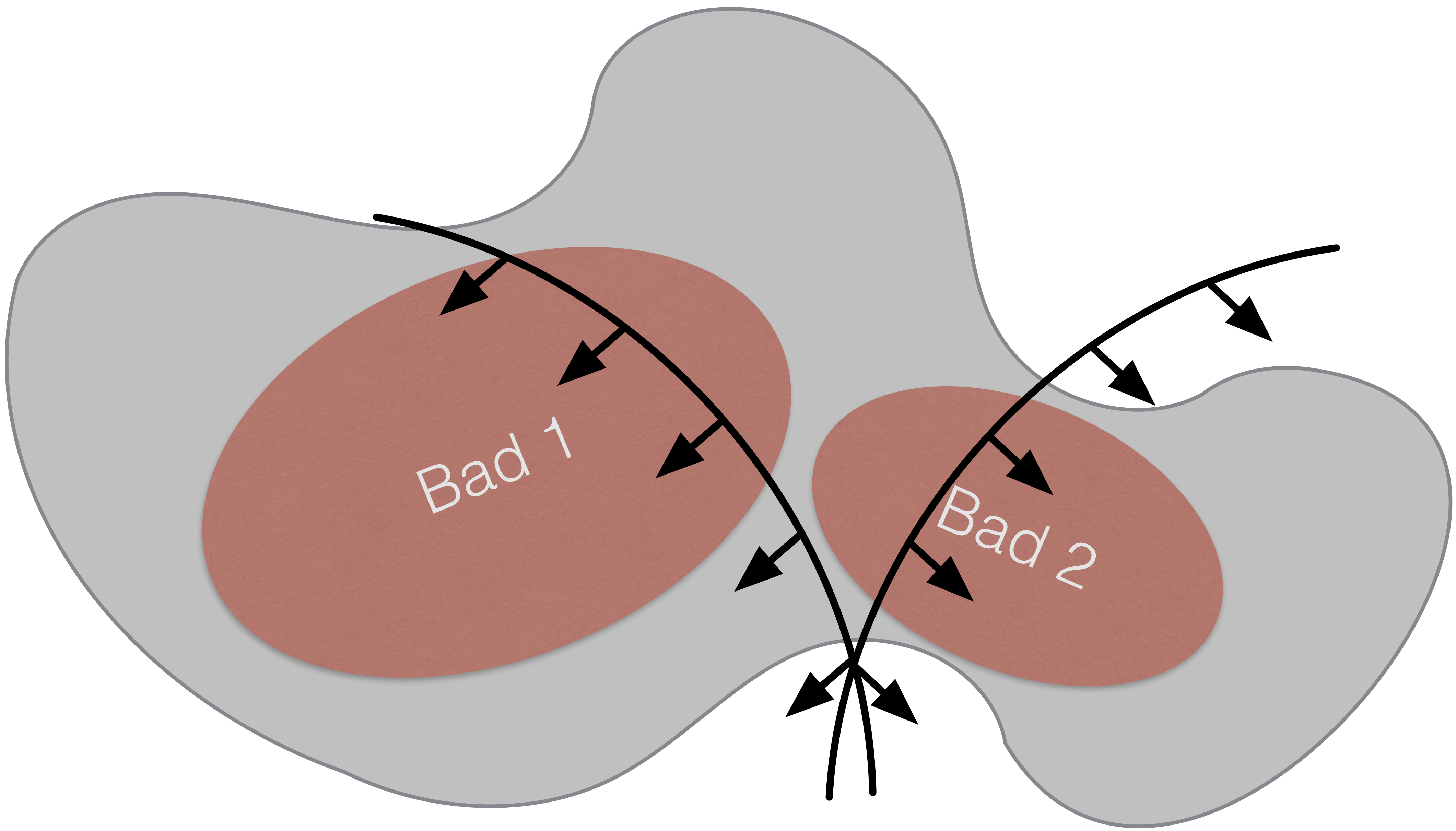
Limits of white magic



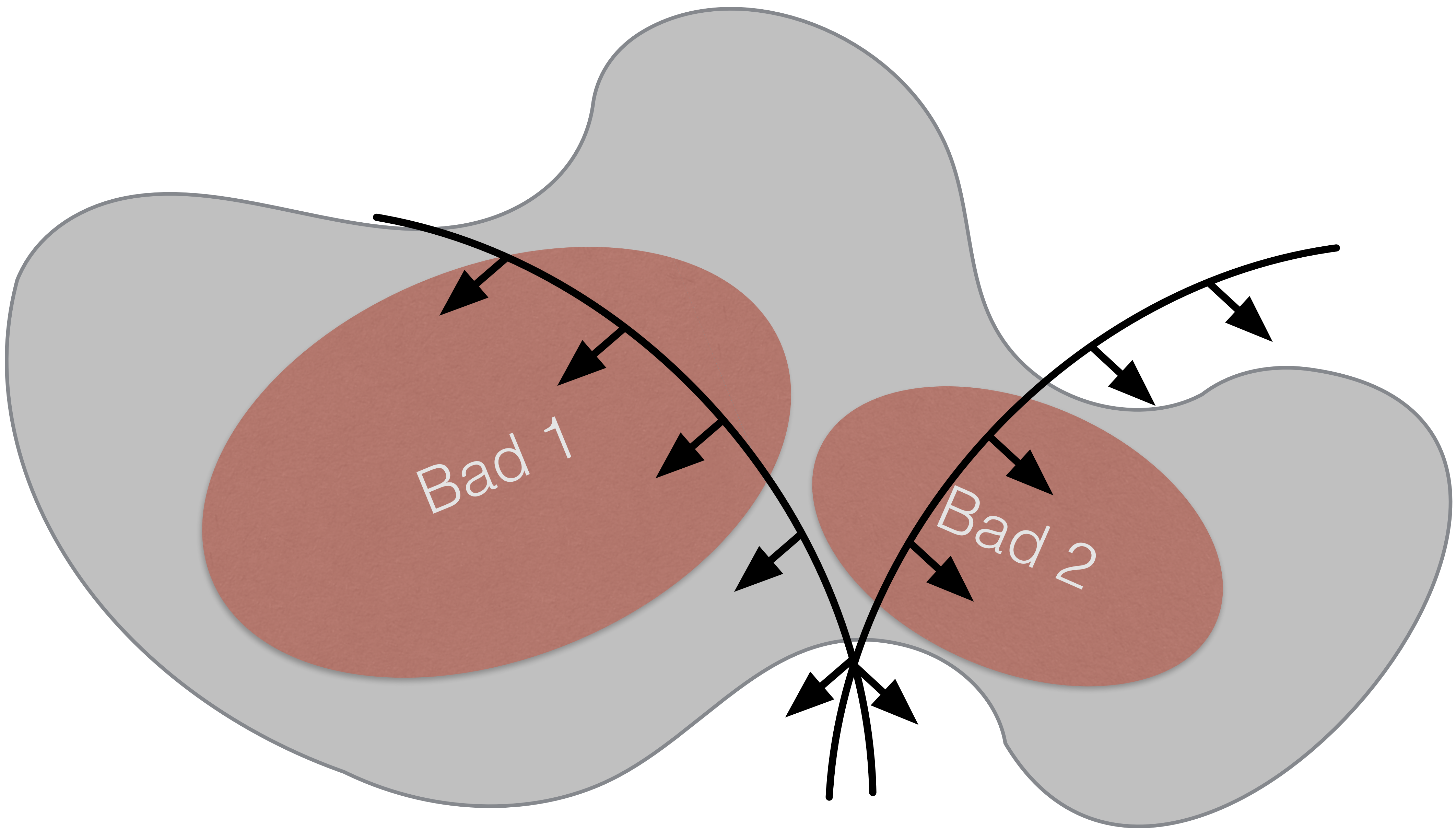
Limits of white magic



Limits of white magic



Limits of white magic



Formally

$$A_1 \oplus A_2 \stackrel{def}{=} (\mathcal{C}_1 \cup \mathcal{C}_2, P_1 \cup P_2, \gamma)$$

$$\begin{aligned}\gamma &\stackrel{def}{=} \left\{a \subseteq 2^P \,\middle|\, a \cap P_1 \in \gamma_1 \ \wedge \ a \cap P_2 \in \gamma_2\right\} \\ &= (\gamma_1 \ltimes P) \cap (\gamma_2 \ltimes P)\end{aligned}$$

Main results: Safety

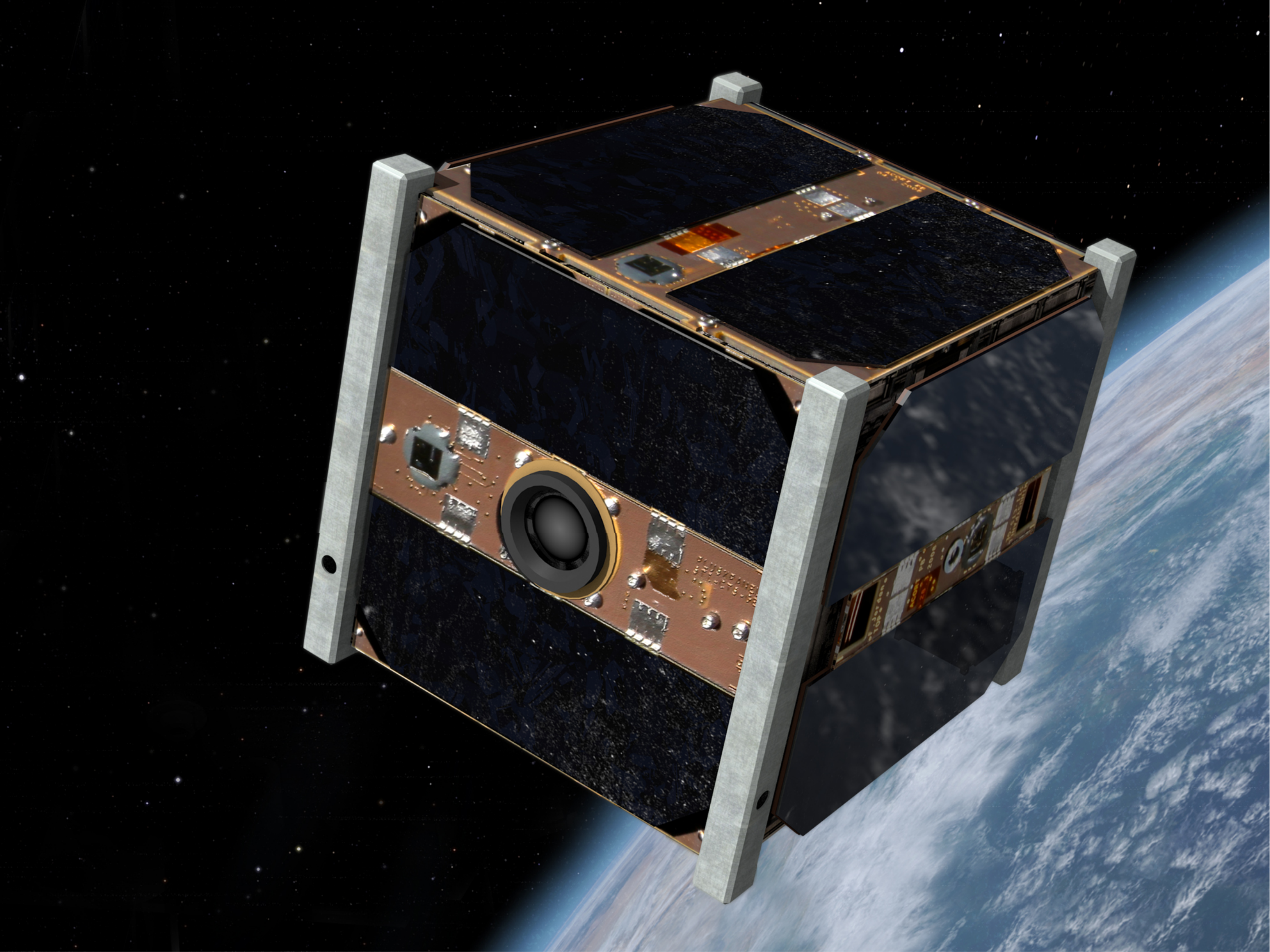
$$\left.\begin{array}{l} A_1(\mathcal{B}) \models \Phi_1 \\ A_2(\mathcal{B}) \models \Phi_2 \end{array}\right\} \quad \Longrightarrow \quad \big(A_1 \oplus A_2\big)(\mathcal{B}) \models \Phi_1 \wedge \Phi_2$$

Safety = *"Bad states never occur"*

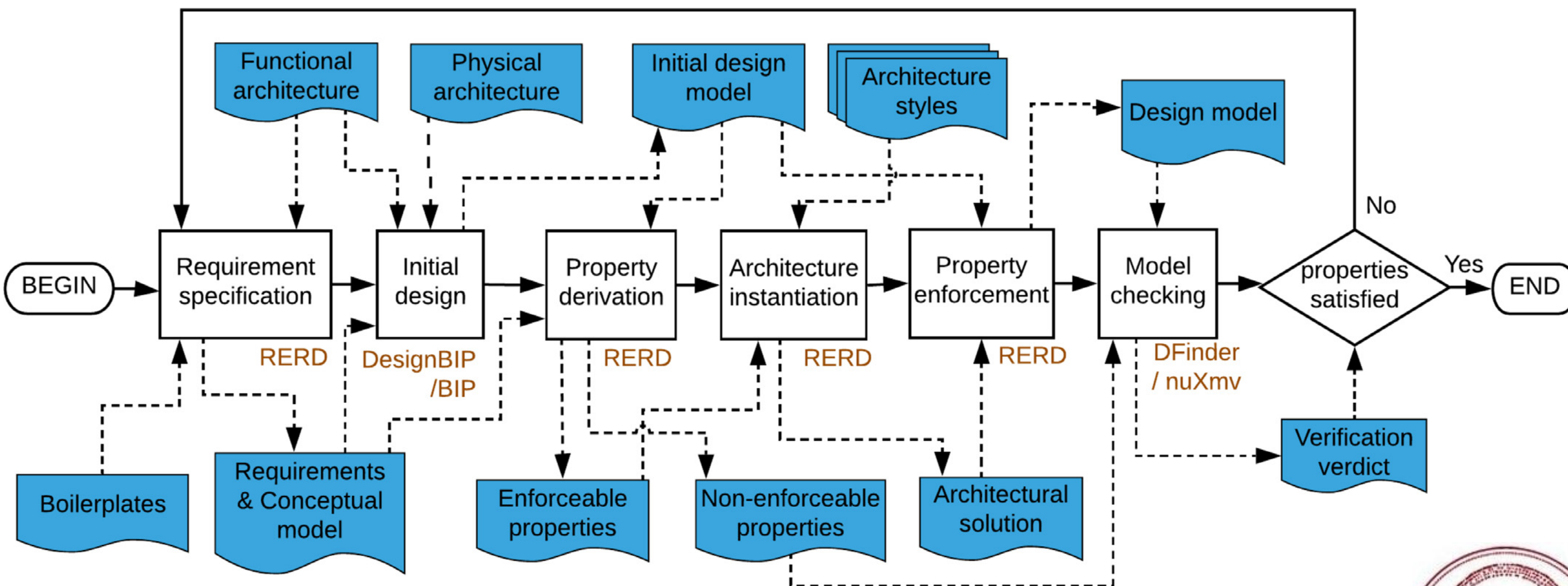
Main results: Liveness

$$\underbrace{\mathcal{A} \quad \left.\begin{array}{r}\text{live}\\ \text{pairwise non-interfering}\end{array}\right\} \quad \Longrightarrow \quad \bigoplus \mathcal{A} \quad \text{live}}_{\text{w.r.t. } \mathcal{B}}$$

Liveness = *"Good states occur infinitely often"*



Requirements and design process



ARISTOTLE UNIVERSITY OF THESSALONIKI

[Stachtiari et al, JSS '18]

CubETH case study

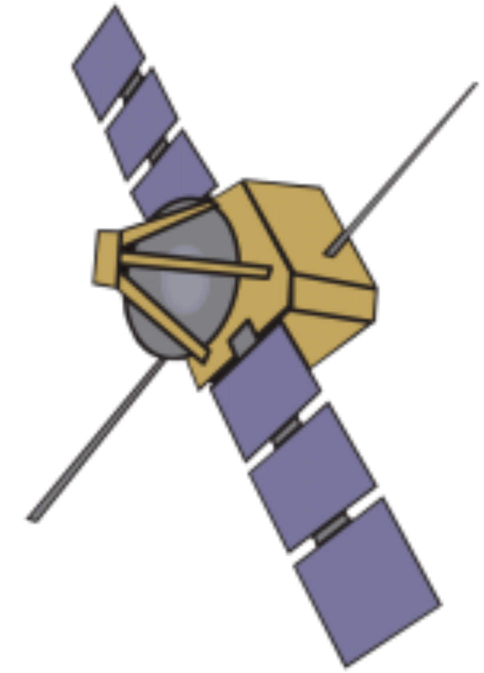
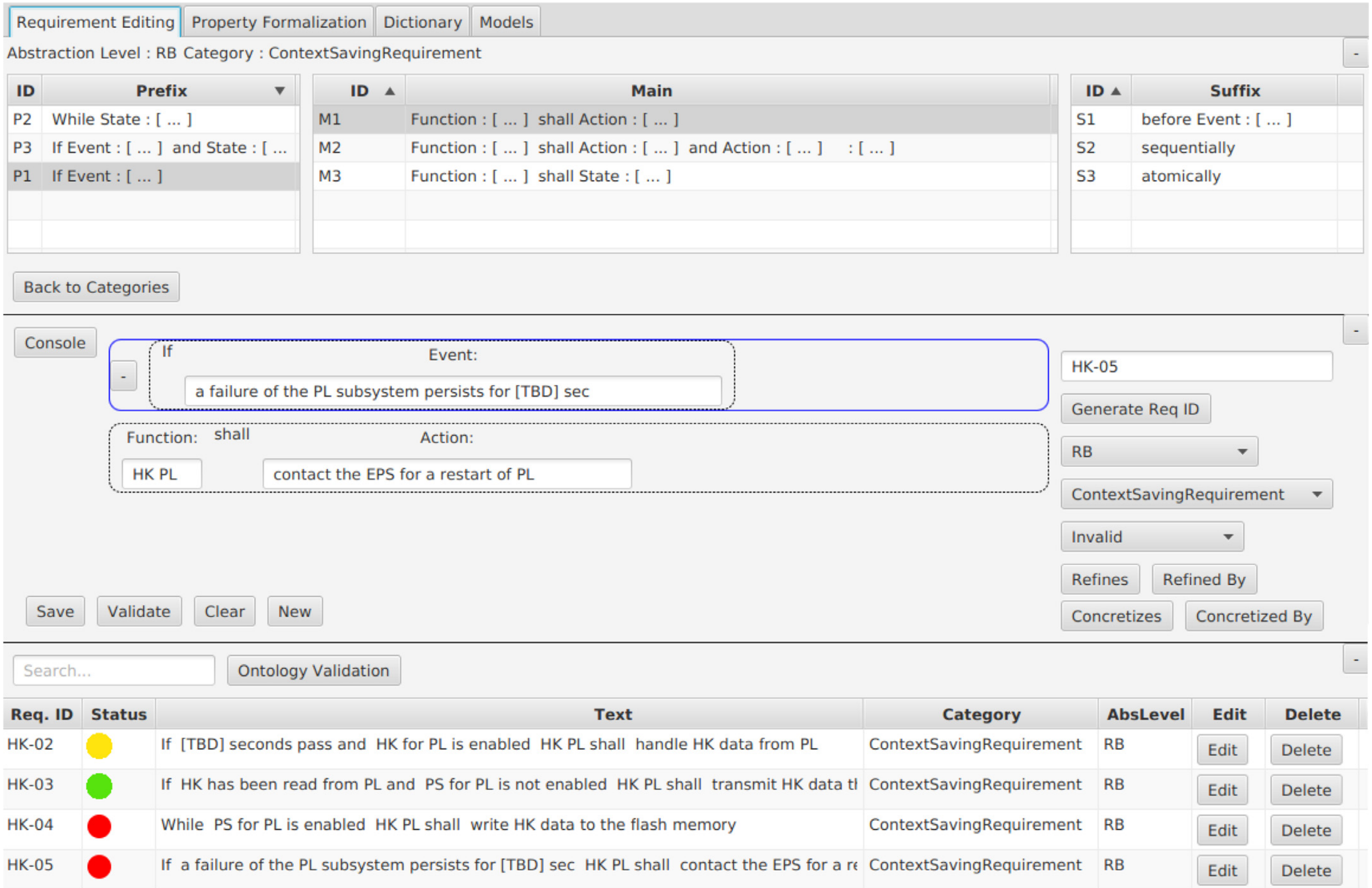


Table 1: Representative requirements for CDMS status and HK_PL

ID	Description
CDMS-007	The CDMS shall periodically reset both the internal and external watchdogs and contact the EPS subsystem with a "heartbeat".
HK-001	The CDMS shall have a Housekeeping activity dedicated to each subsystem.
HK-003	When line-of-sight communication is possible, housekeeping information shall be transmitted through the COM subsystem.
HK-004	When line-of-sight communication is not possible, housekeeping information shall be written to the non-volatile flash memory.
HK-005	A Housekeeping subsystem shall have the following states: NOMINAL, ANOMALY and CRITICAL_FAILURE.

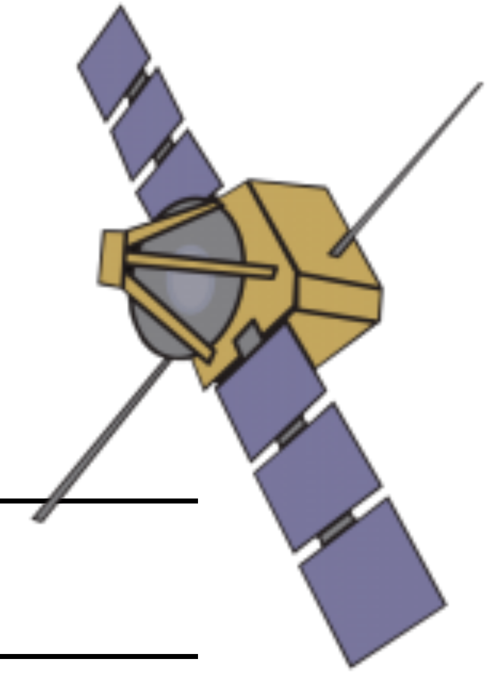
[Mavridou et al, FACS '16]

RERD tool



Requirement Editing
Property Formalization
Dictionary
Models
Abstraction Level : RB Category : ContextSavingRequirement
ID
Prefix
P2 While State : [...]
P3 If Event : [...] and State : [...
P1 If Event : [...]
ID
Main
M1 Function : [...] shall Action : [...]
M2 Function : [...] shall Action : [...] and Action : [...] : [...]
M3 Function : [...] shall State : [...]
ID
Suffix
S1 before Event : [...]
S2 sequentially
S3 atomically
Back to Categories
Console
If
Event:
a failure of the PL subsystem persists for [TBD] sec
Function:
shall
Action:
HK PL
contact the EPS for a restart of PL
HK-05
Generate Req ID
RB
ContextSavingRequirement
Invalid
Refines
Refined By
Concretizes
Concretized By
Save
Validate
Clear
New
Search...
Ontology Validation
Req. ID
Status
Text
Category
AbsLevel
Edit
Delete
HK-02 If [TBD] seconds pass and HK for PL is enabled HK PL shall handle HK data from PL ContextSavingRequirement RB Edit Delete
HK-03 If HK has been read from PL and PS for PL is not enabled HK PL shall transmit HK data tl ContextSavingRequirement RB Edit Delete
HK-04 While PS for PL is enabled HK PL shall write HK data to the flash memory ContextSavingRequirement RB Edit Delete
HK-05 If a failure of the PL subsystem persists for [TBD] sec HK PL shall contact the EPS for a re ContextSavingRequirement RB Edit Delete

CubETH case study

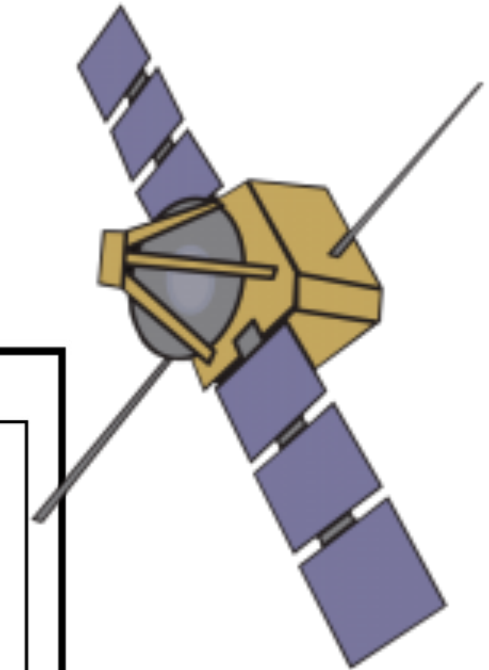


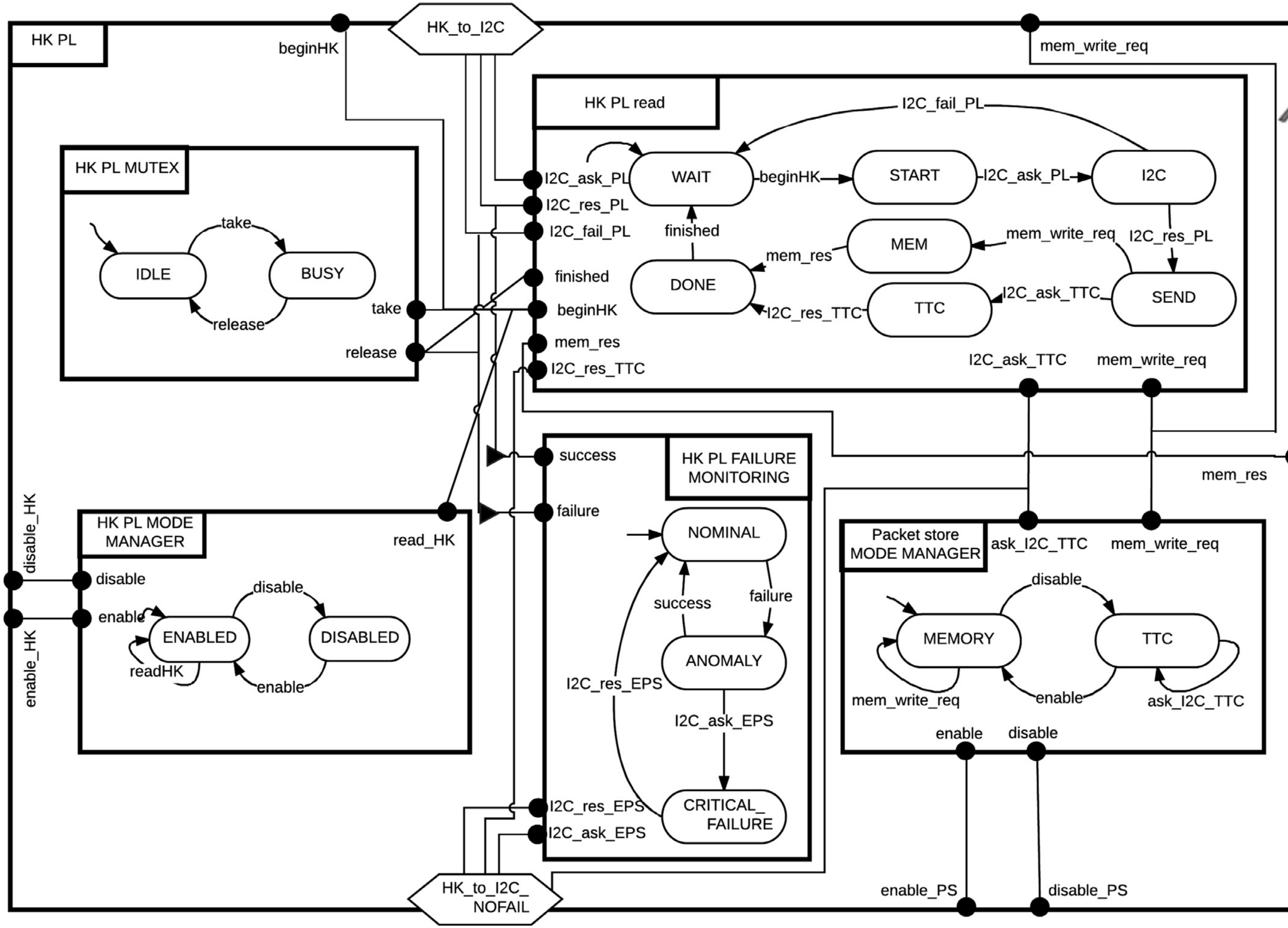
Requirements for the *HK PL* function.

ID	Requirement
HK-02	P2: if ⟨event-e003: [TBD] sec pass ⟩ and ⟨state-s003: HK collection is enabled for PL ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a004: handle HK data from the PL ⟩
HK-03	P3: if ⟨state-s002: PS[a] for PL is not enabled ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a002: transmit HK data through the TC/TM service ⟩
HK-04	P3: while ⟨state-s001: PS for PL is enabled ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a001: write HK data to the flash memory ⟩
HK-05	P1: if ⟨event-e004: a PL failure persists for [TBD] sec ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a003: contact the EPS for a restart of the PL ⟩

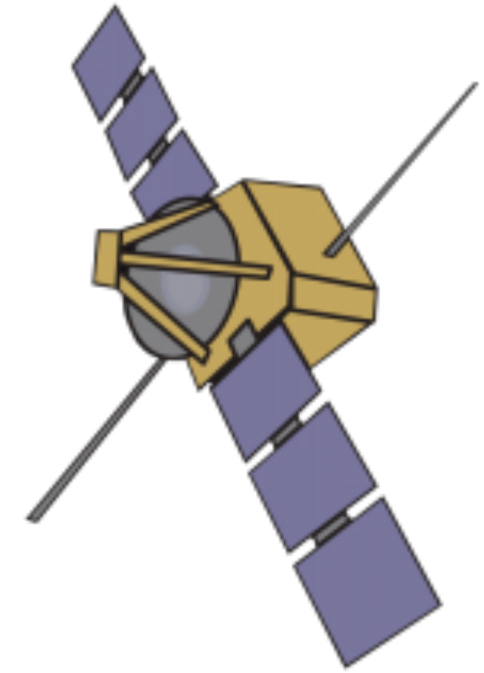
[a] PS stands for a packet store structure.

CubETH case study





CubETH case study



Durations and input sizes of the process steps.

Step	Duration	Input size
Requirement specification	8 h	38 requirements
Initial design	5 h	12 components
Architecture instantiation	3 h	47 enforced properties
Verification of deadlock freedom	12 s	46 components

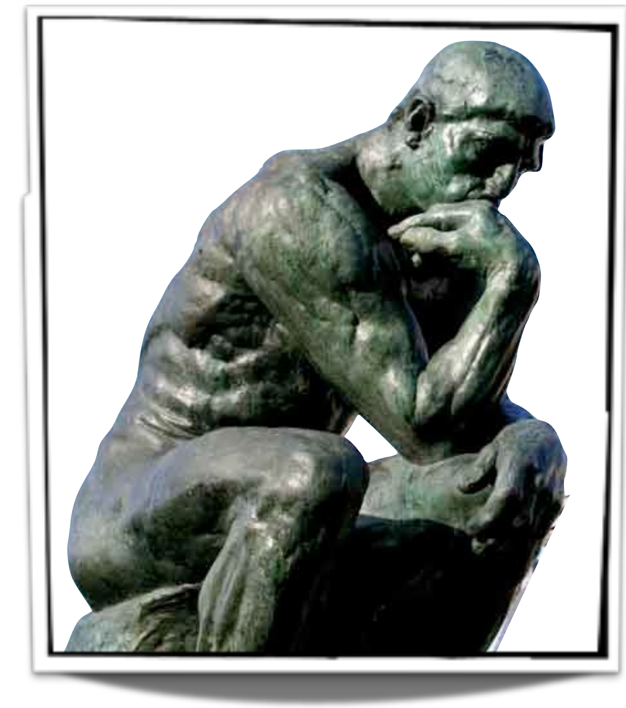
Statistics of requirement formulation and property enforcement.

Model	Flow	Mode	Event	Mutex	Failure	Requir.	Deriv. Prop.	Assum. Prop.	Enforced
Payload	0	2	0	4	0	12	16	0	16
HK PL	0	2	1	1	1	4	6	0	6
HK EPS	0	2	1	1	1	4	6	0	6
HK COM	0	2	1	1	1	4	6	0	6
HK CDMS	0	2	1	1	0	3	4	0	4
Flash memory	0	1	0	1	0	8	13	4	3
CDMS status	1	0	0	0	0	1	3	0	3
Error logging	0	0	1	1	0	2	3	0	3
Total	1	11	5	10	3	38	57	4	47

Summary

Mastering system complexity requires

- Manipulating models to raise the abstraction level
- Expressive enough to avoid ad-hoc solutions
- Simple enough to be acceptable for engineers



Rigorous design workflow

- Validate first, then generate the code
- A sequence of semantics-preserving transformations

Further information

FOCUS: SOFTWARE COMPONENTS: BEYOND PROGRAMMING

Rigorous Component-Based System Design Using the BIP Framework

Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis,
Verimag Laboratory

// An autonomous robot case study illustrates the use of the behavior, interaction, priority (BIP) component framework as a unifying semantic model to ensure correctness of essential system design properties. //

SYSTEM DESIGN DIFFERS radically from pure software design in that it must account not only for functional requirements but also for extrafunctional requirements regarding the use of execution platform resources, such as time, memory, and energy. Meeting extrafunctional requirements is essential in embedded system design and requires evaluation of how design choices affect overall system behavior. It also implies a deep understanding of how the application software interacts with the underlying execution platform. Yet system designers currently lack rigorous techniques for deriving global models of a given system from models of its application software and execution platform.

We define a rigorous design flow as one that guarantees essential system properties. Most existing design flows that aspire to this goal privilege a unique programming model and associate it with a compilation chain that's adapted for a given execution model. For example, synchronous system design relies on synchronous programming models and usually targets hardware or sequential implementations on single processors.[1] Alternatively, real-time programming, based on scheduling theory for periodic tasks, targets dedicated real-time multitasking platforms.[2]

At the Verimag Laboratory, we've been developing the behavior, interaction, priority (BIP) component framework to support a rigorous system design flow. The BIP framework is

- *model-based*, describing all software and systems according to a single semantic model. This maintains the flow's overall coherency by guaranteeing that a description at step *n+1* meets essential properties of a description at step *n*.
- *component-based*, providing a family of operators for building composite components from simpler components. This overcomes the poor expressiveness of theoretical frameworks based on a single operator, such as the product of automata or a function call.
- *tractable*, guaranteeing correctness

The Journal of Systems & Software 145 (2018) 52–78
Contents lists available at ScienceDirect
The Journal of Systems & Software
journal homepage: www.elsevier.com/locate/jss

ELSEVIER

Early validation of system requirements and design through correctness-by-construction

Emmanouela Stachtiari[*,a], Anastasia Mavridou[b], Panagiotis Katsaros[a], Simon Bliudze[c], Joseph Sifakis[d]

[a] *Department of Informatics, Aristotle University of Thessaloniki, Greece*
[b] *Robust Software Engineering Group SGT Inc., NASA Ames Research Center Moffett Field, CA, USA*
[c] *INRIA Lille –Europe, Nord , France*
[d] *Verimag, Université Grenoble Alpes, France*

ARTICLE INFO

Keywords:
Rigorous system design
Requirements formalization
Model-based design
Correctness-by-construction

ABSTRACT

Early validation of requirements aims to reduce the need for the high-cost validation testing and corrective measures at late development stages. This work introduces a systematic process for the unambiguous specification of system requirements and the guided derivation of formal properties, which should be implied by the system 's structure and behavior in conjunction with its external stimuli. This rigorous design takes place through the incremental construction of a model using the BIP (Behavior-Interaction-Priorities) component framework. It allows building complex designs by composing simpler reusable designs enforcing given properties. If some properties are neither enforced nor verified, the model is refined or certain requirements are revised. A validated model provides evidence of requirements' consistency and design correctness. The process is semi-automated through a new tool and existing verification tools. Its effectiveness was evaluated on a set of requirements for the control software of the CubETH nanosatellite and an extract of software requirements for a Low Earth Orbit observation satellite. Our experience and obtained results helped in identifying open challenges for applying the method in industrial context. These challenges concern with the domain knowledge representation, the expressiveness of used specification languages, the library of reusable designs and scalability.

1. Introduction

1.1. Problem statement

The design problem in systems engineering concerns with defining the architecture, modules, interfaces and data for a system, in order to meet given requirements (Buede and Miller, 2016). Initially, requirements are high-level statements (conditions or capabilities that are also called stakeholder requirements) (Fuxman et al., 2004), from which the system requirements are derived that define what the system must do to satisfy stakeholder requirements (Hull et al., 2010). In this article, we focus specifically on system requirements; when we refer to stakeholder requirements we do so explicitly.

In Sifakis (2013) and Benveniste et al. (2015), two perspectives of *rigorous system design* are introduced. The term "rigorous" refers to a formal model-based process that leads from requirements to correct implementations. In particular, the focus is on the design problem for systems that continuously interact with an external environment; such systems usually involve concurrent execution and emergent behaviors. The design process can be decomposed into two phases. During the first phase, which is called *proceduralization* in Sifakis (2013), the declarative system requirements are transformed into a procedure, i.e., a model prescribing how the anticipated functionality can be realized by executing sequences of elementary functions. During the second phase, which is called *materialization*, the procedure is implemented in a system that meets all extra-functional requirements by using available resources cost-effectively.

In this article, we introduce a *model-based* approach for the proceduralization phase, which aims to the systematic development of a design solution for a set of system requirements. The design problem is well-defined, only if the requirements fulfill essential properties, i.e., if they are complete, consistent, correct (valid for an acceptable solution), and attainable. However, requirements provide in principle only a partial specification, which according to the current industrial practice

* Corresponding author.
E-mail addresses: emmastac@csd.auth.gr (E. Stachtiari), anastasia.mavridou@nasa.gov (A. Mavridou), katsaros@csd.auth.gr (P. Katsaros), simon.bliudze@inria.fr (S. Bliudze), Joseph.Sifakis@univ-grenoble-alpes.fr (J. Sifakis).

https://doi.org/10.1016/j.jss.2018.07.053
Received 26 September 2017; Received in revised form 14 March 2018; Accepted 17 July 2018
Available online 20 July 2018