



Mixed Criticality

Cost-Effective Demonstration of non-Interference

Praxis High Integrity Systems



Agenda

- The segregation problem
- Preconditions for analysis
- Information flow analysis
- SPARK extensions
- Example
- Conclusions

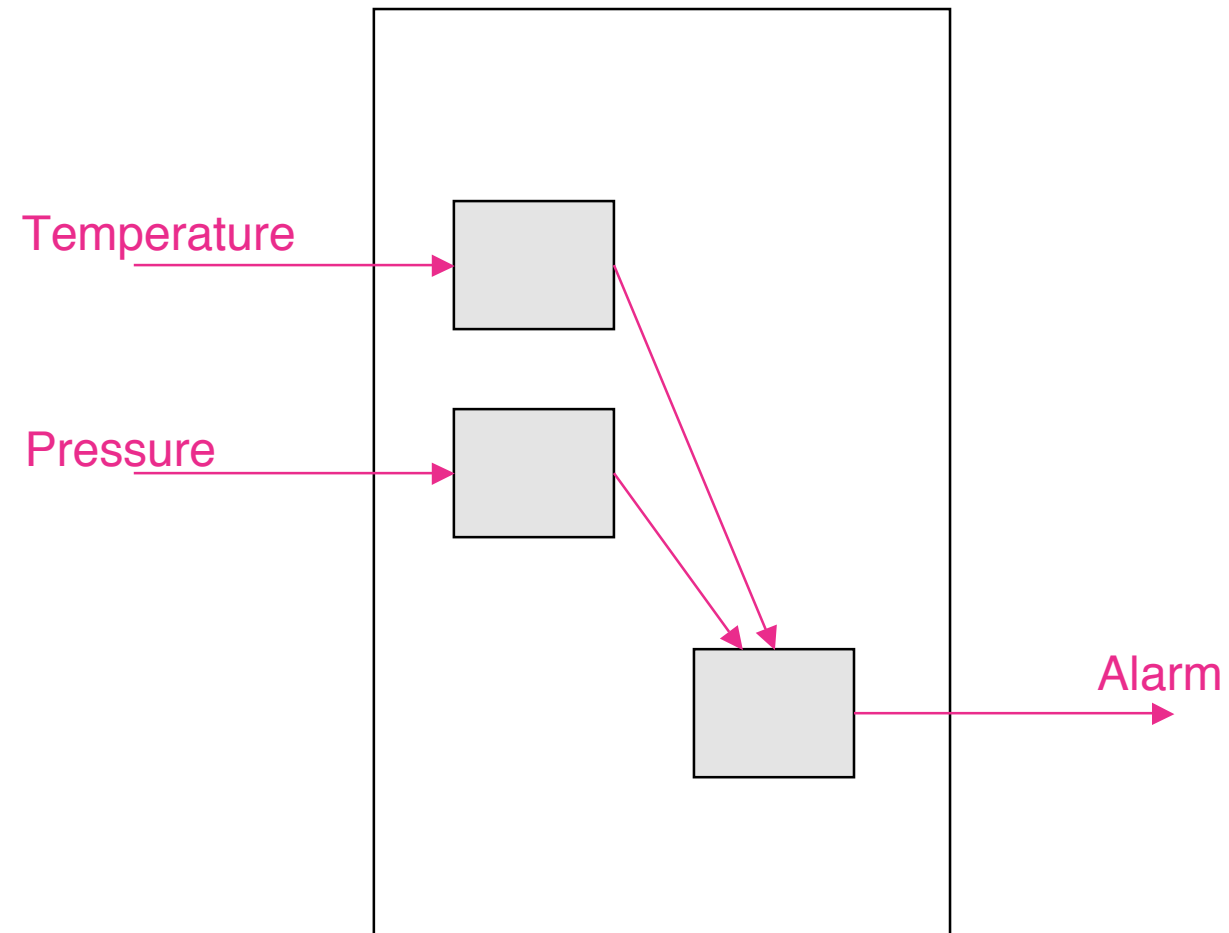


Agenda

- The segregation problem
- Preconditions for analysis
- Information flow analysis
- SPARK extensions
- Example
- Conclusions

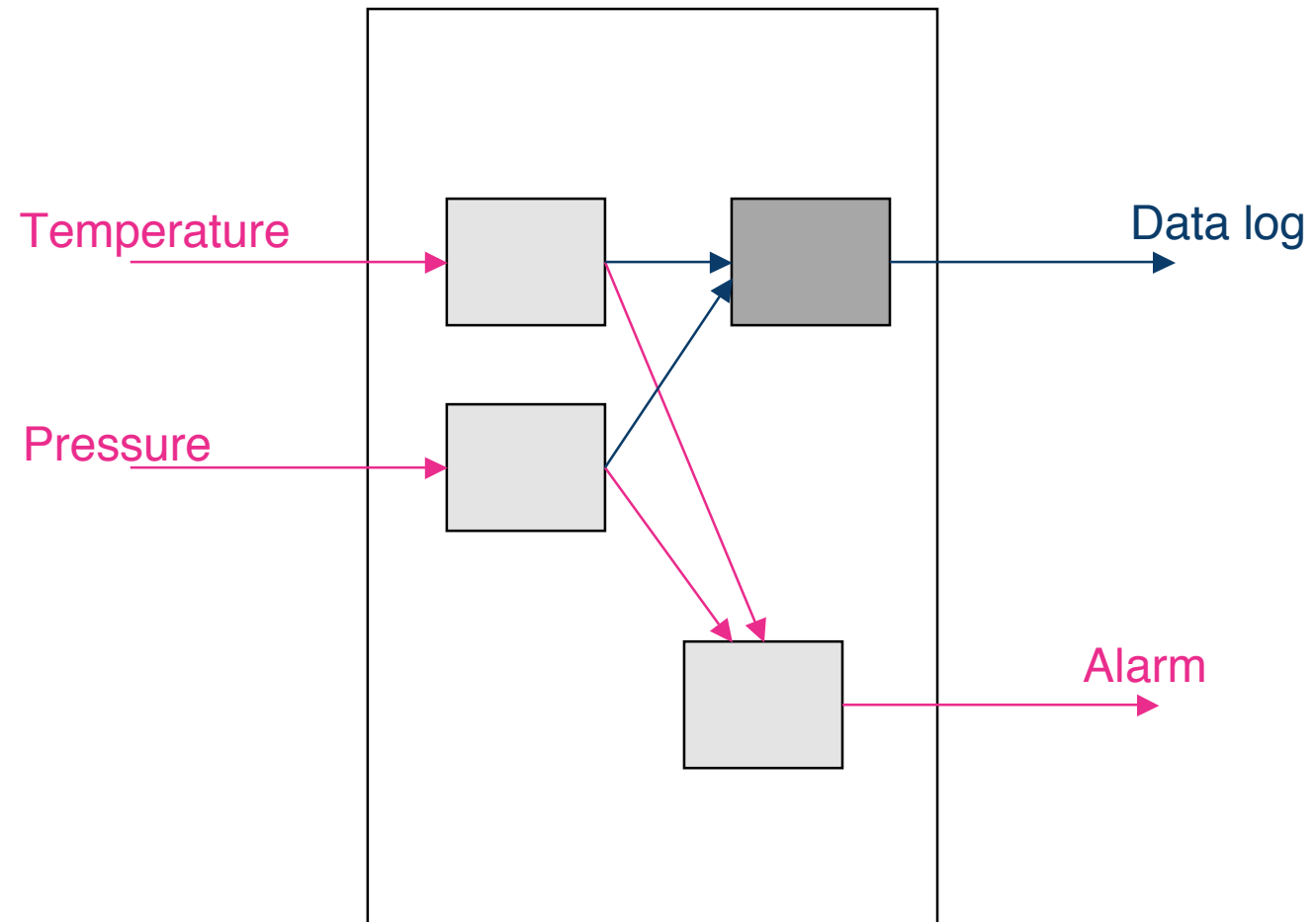


Segregation problem





Segregation problem





Agenda

- The segregation problem
- Preconditions for analysis
- Information flow analysis
- SPARK extensions
- Example
- Conclusions



Preconditions

- A logically sound environment
 - no uninitialized variables
 - no memory corruption via array bounds or pointers
 - no run-time errors
 - no deadlocks or other resource hogging
- SPARK provides the necessary foundations: e.g.
 - data flow analysis
 - proof of absence of run-time errors
 - Ravenscar profile



Agenda

- The segregation problem
- Preconditions for analysis
- **Information flow analysis**
- SPARK extensions
- Example
- Conclusions

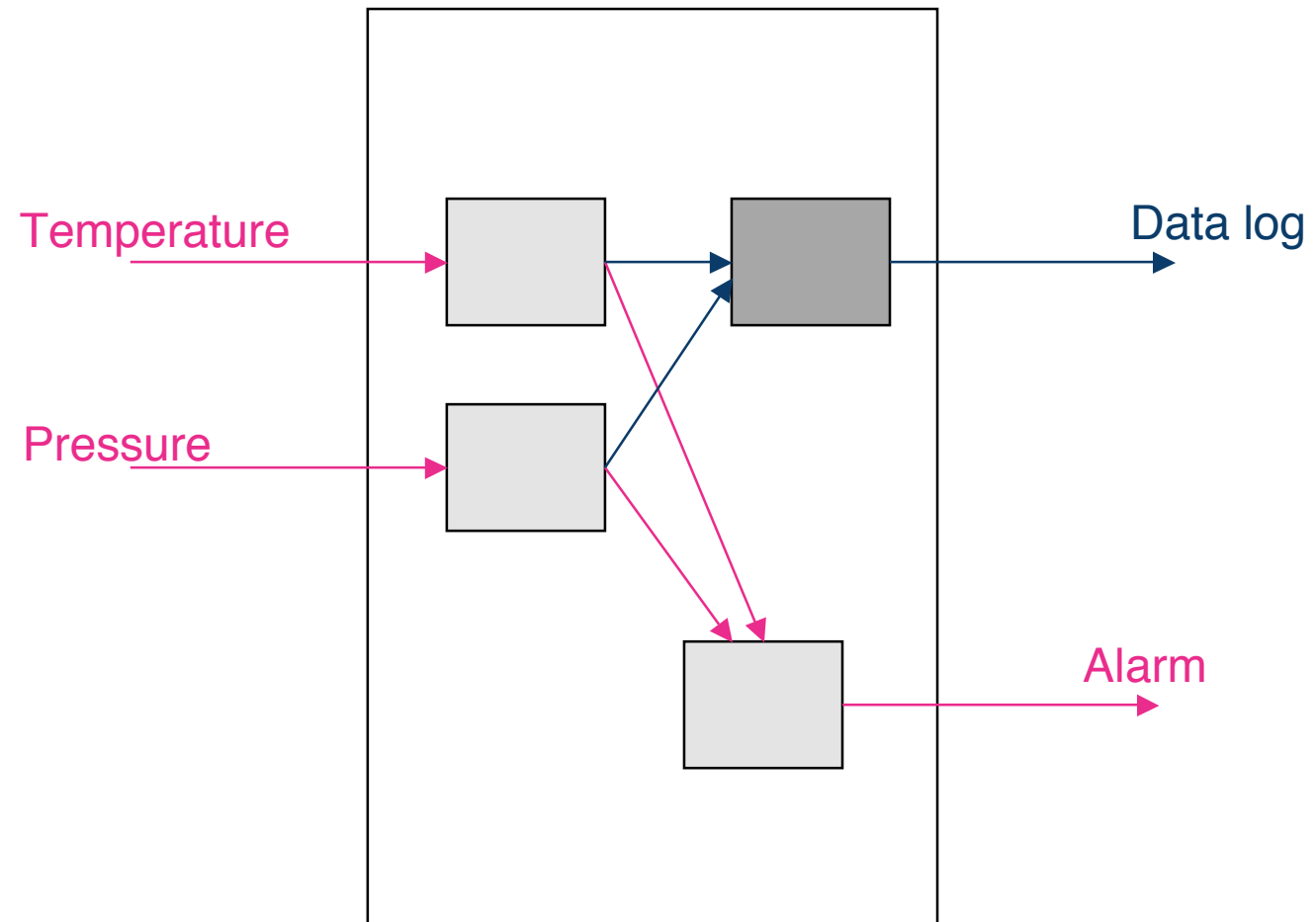


Information flow analysis

- IFA determines **influence** of one variable on another through:
 - assignment
 - effect on control flow
- IFA is the technique that lets us show separation
- Foundation work:
 - overall system information flow
 - reported at SIGAda 2004

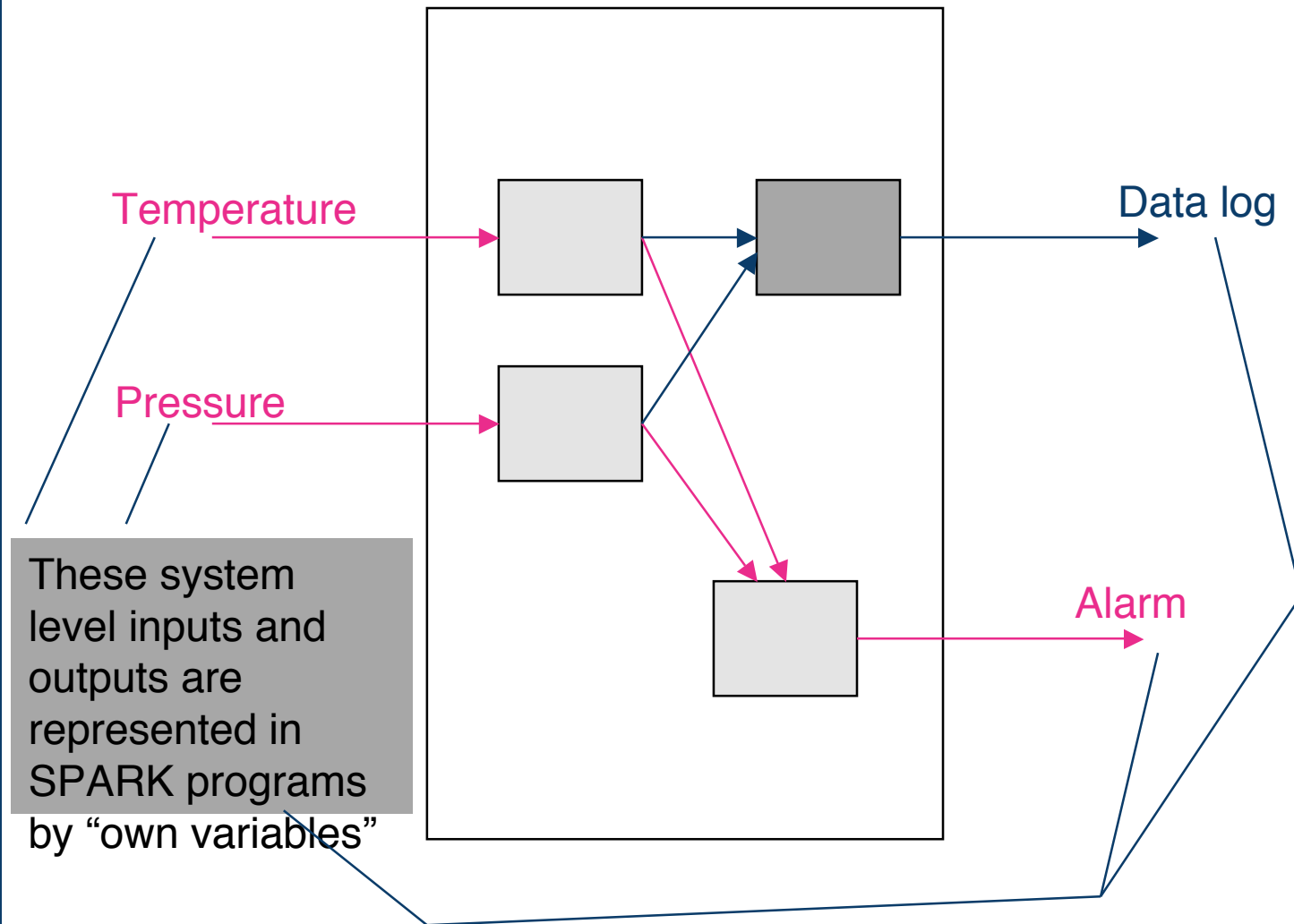


Overall system information flow





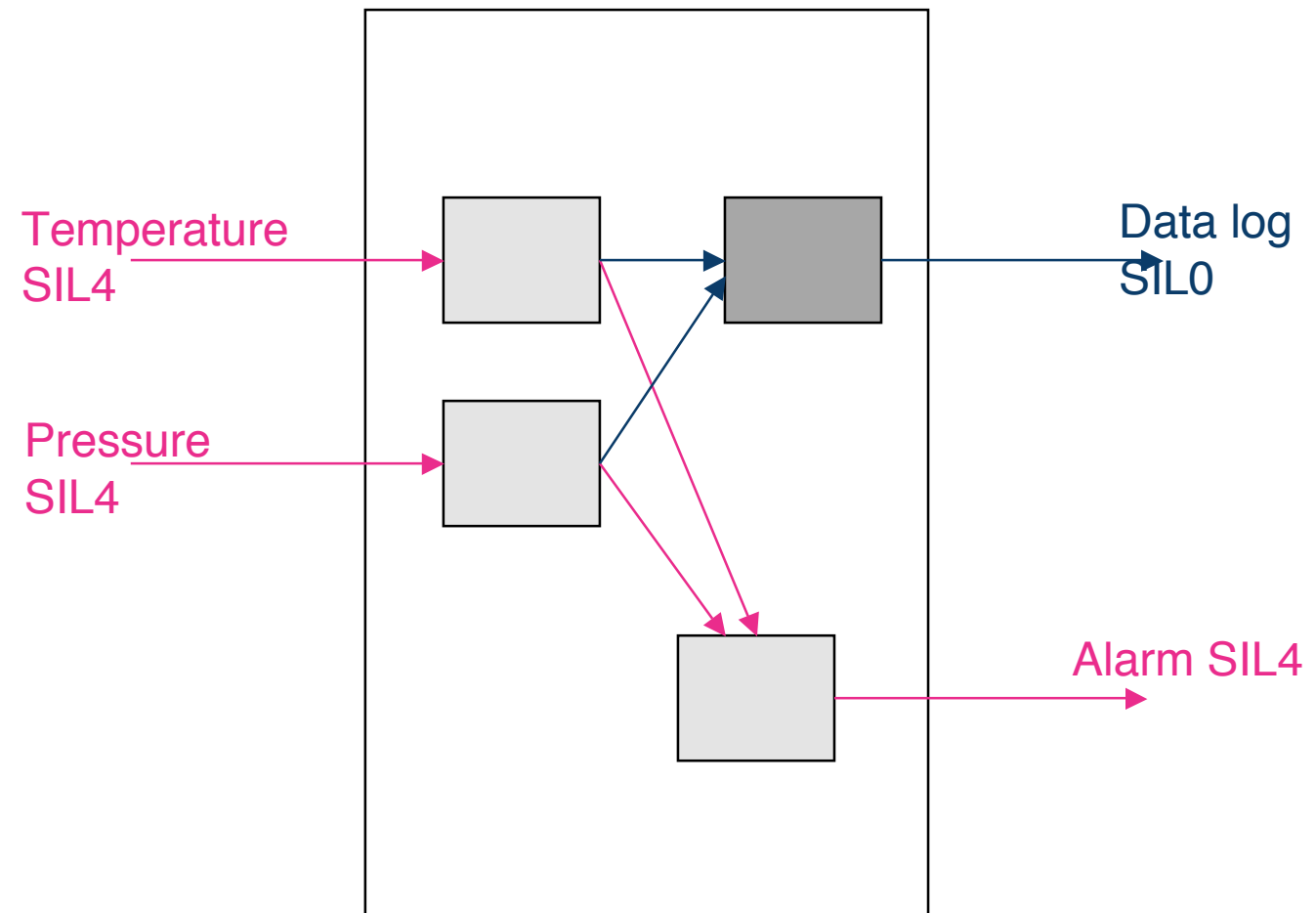
Overall system information flow



These system level inputs and outputs are represented in SPARK programs by "own variables"

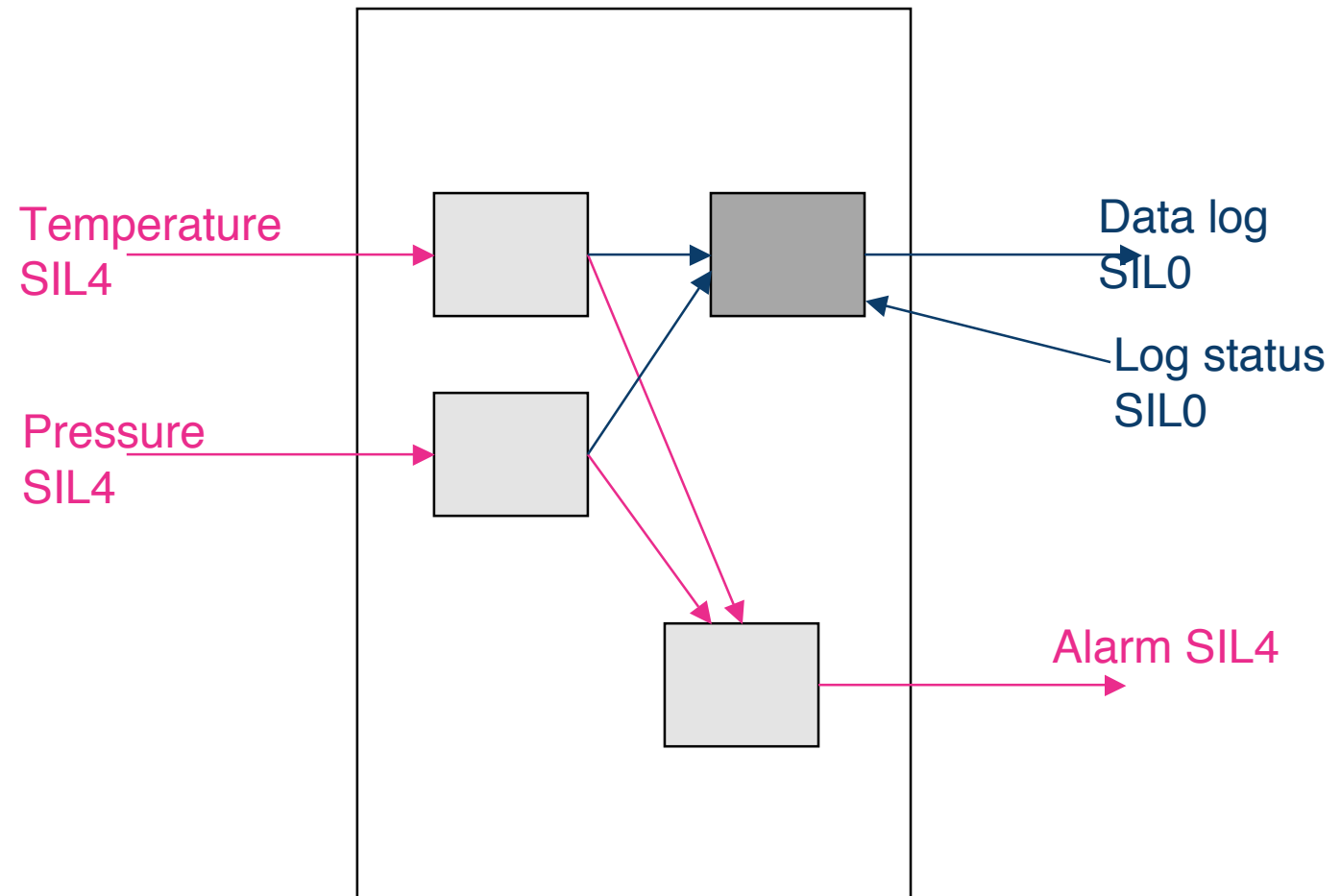


Overall system information flow



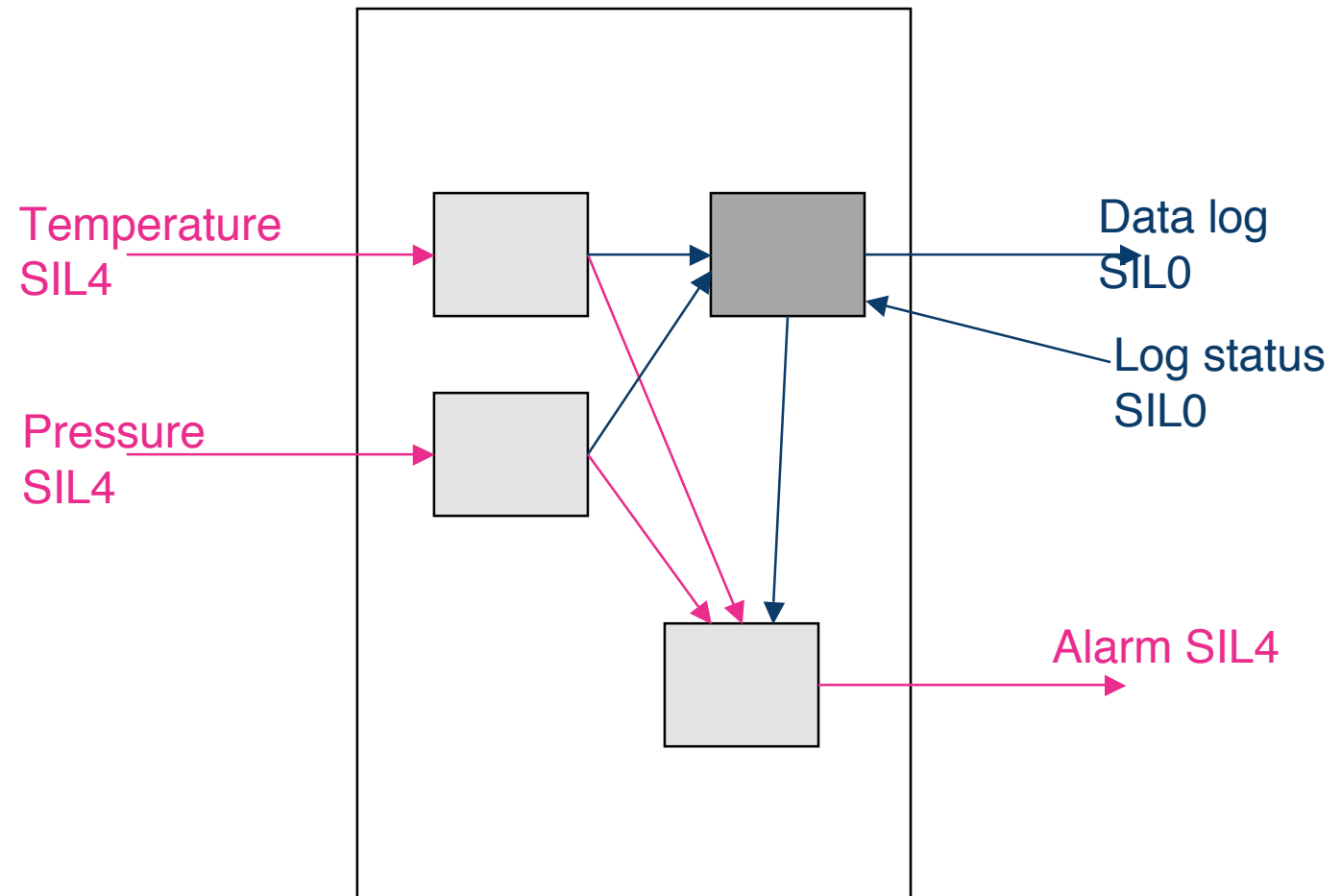


Overall system information flow



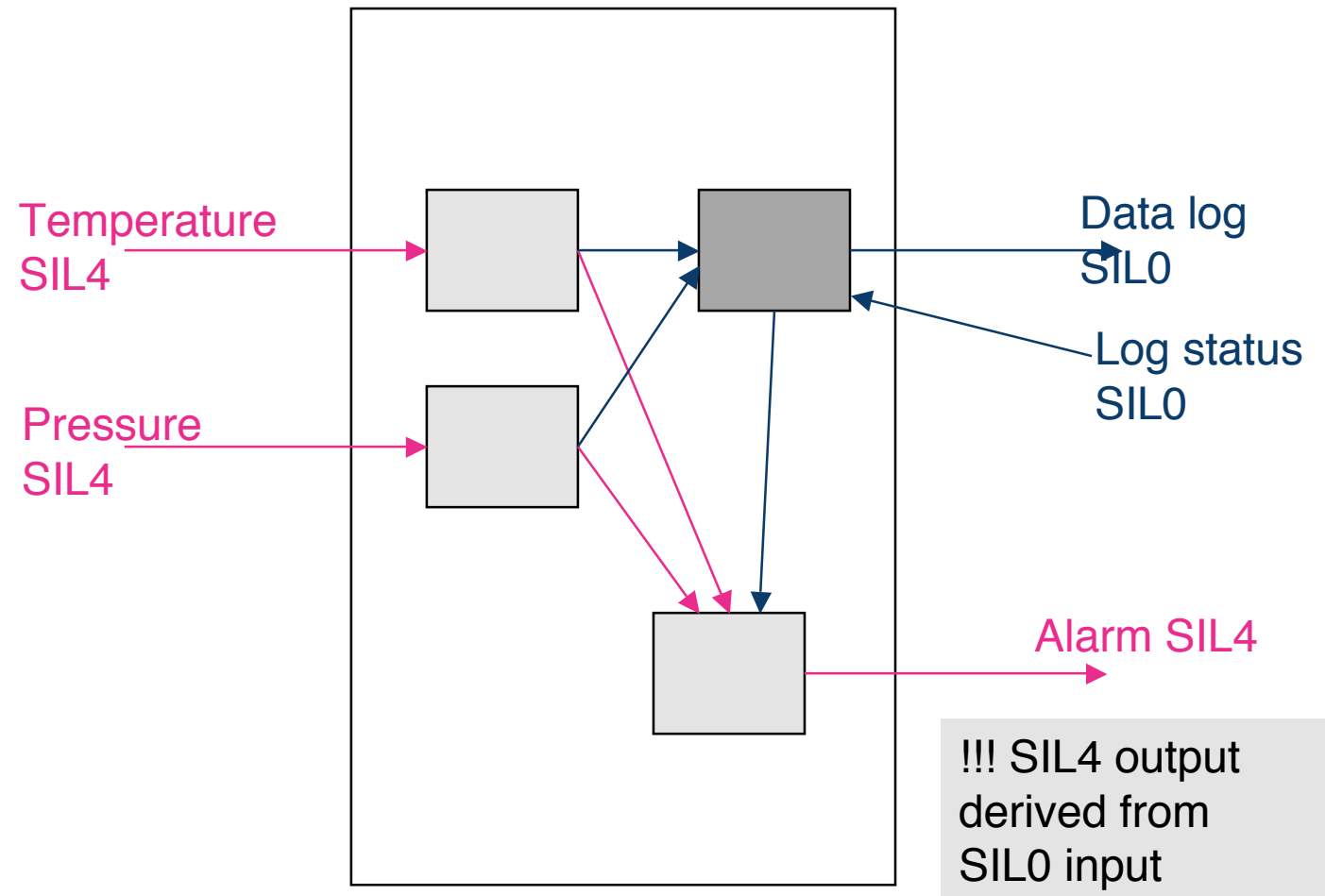


Overall system information flow





Overall system information flow





Agenda

- The segregation problem
- Preconditions for analysis
- Information flow analysis
- **SPARK extensions**
- Example
- Conclusions

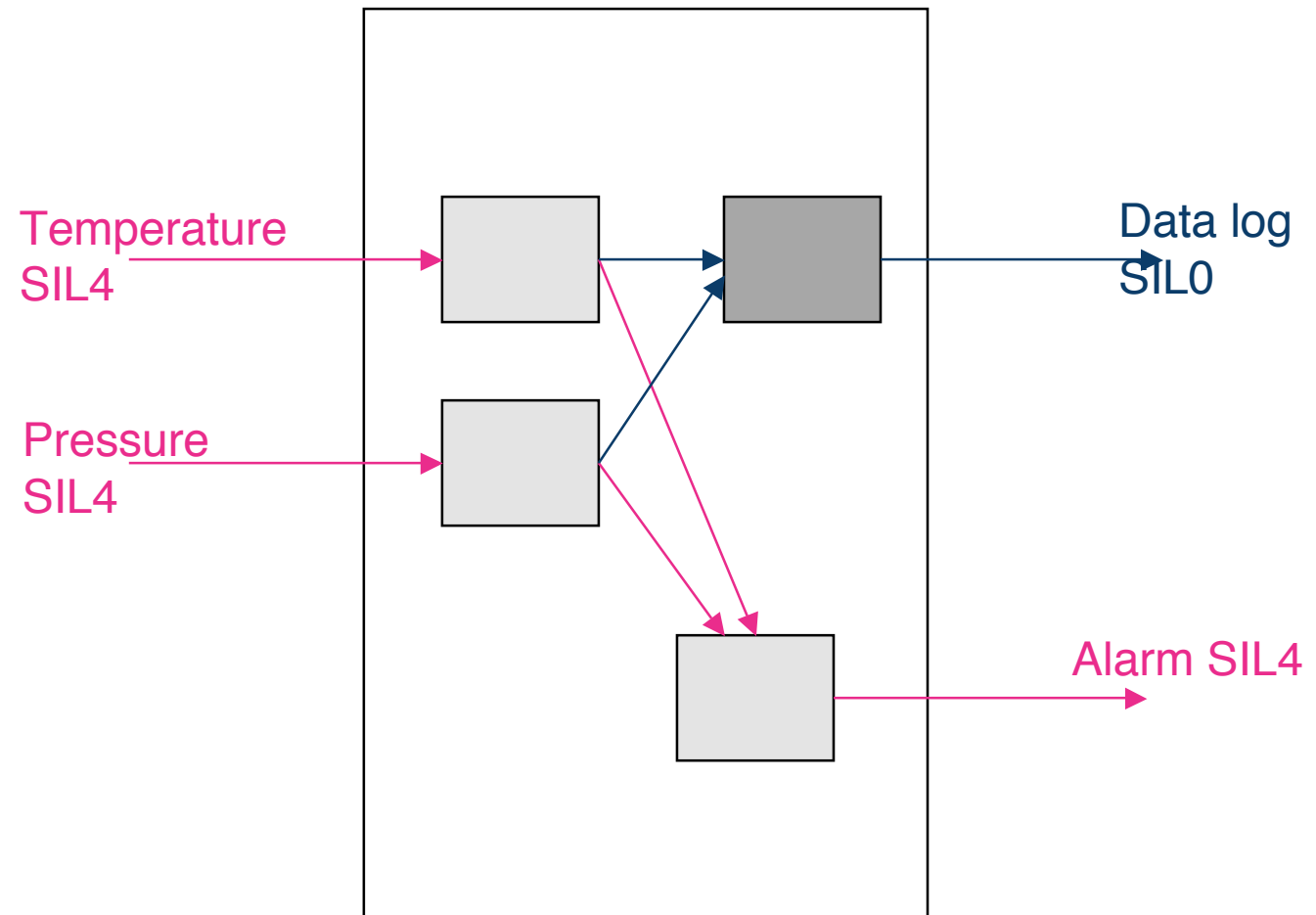


SPARK Extensions

- End-to-end information flow is only part of the solution
- We want to identify which **statement sequences** are involved in critical activities
- We don't want to wait until the program is complete
- Solution:
 - use annotations to **assert** intended criticality of each subprogram
 - checked by the Examiner
 - **the maths** is in our Ada Europe 2005 paper!

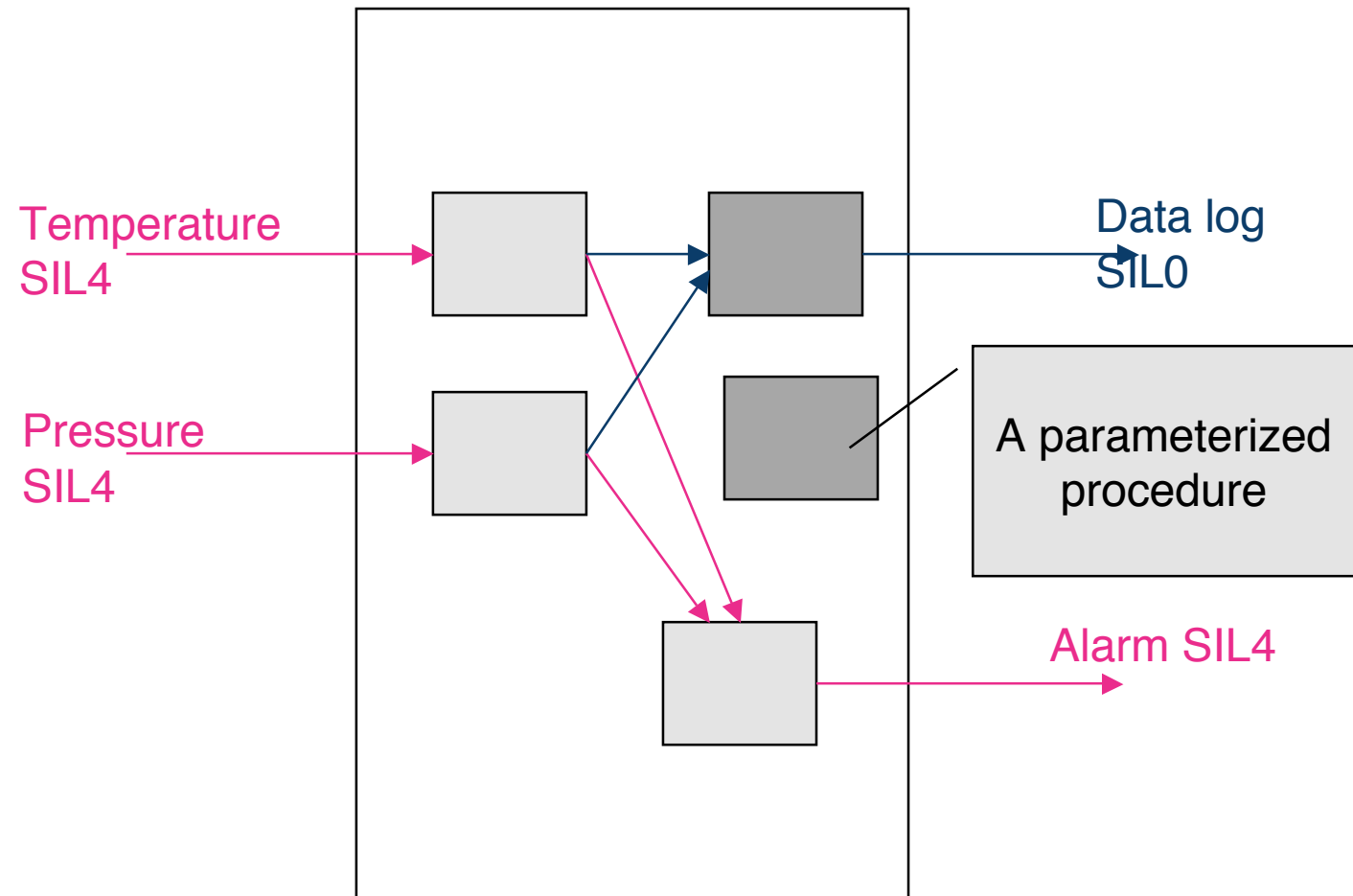


Subprogram level information flow



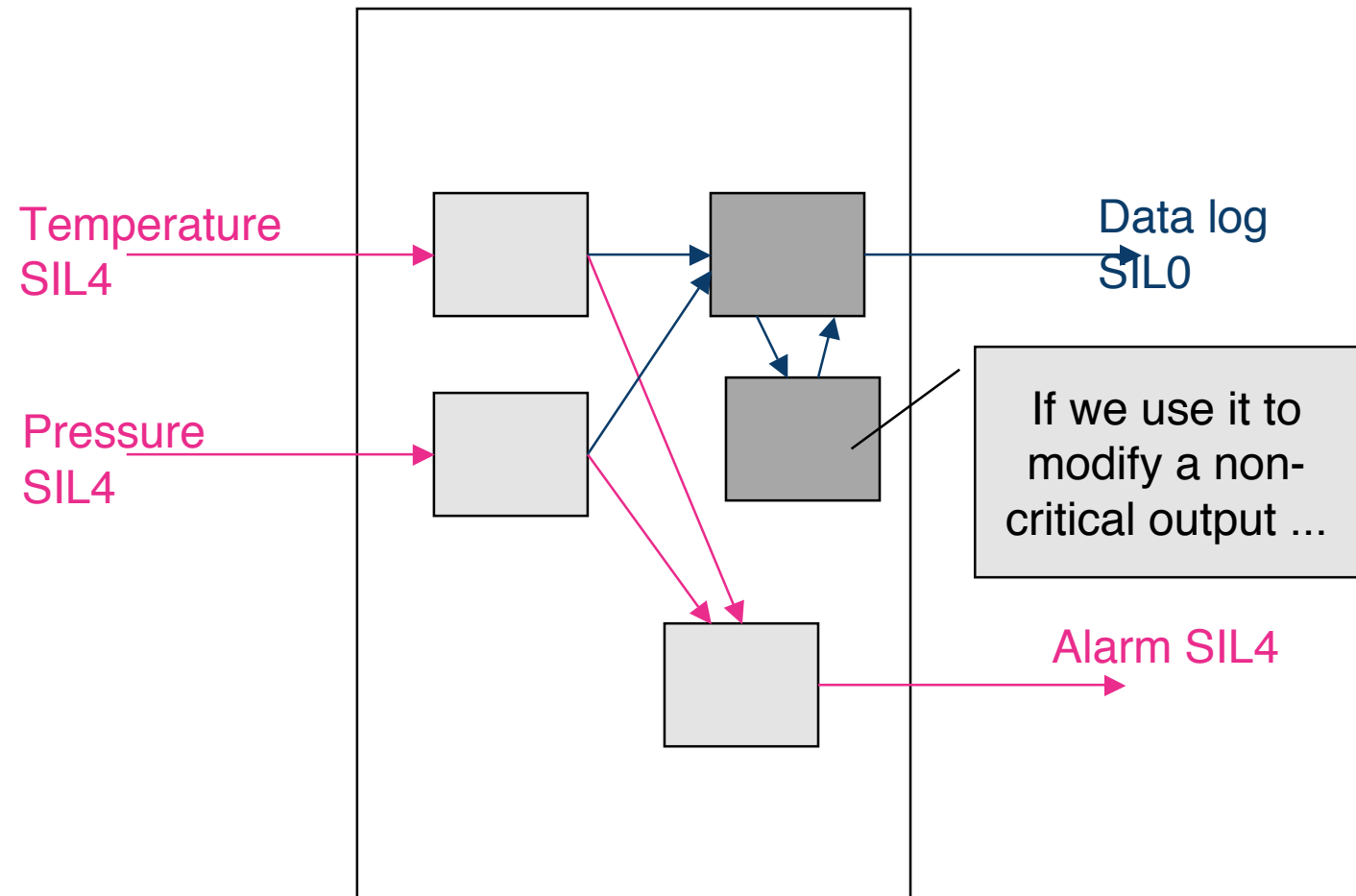


Subprogram level information flow



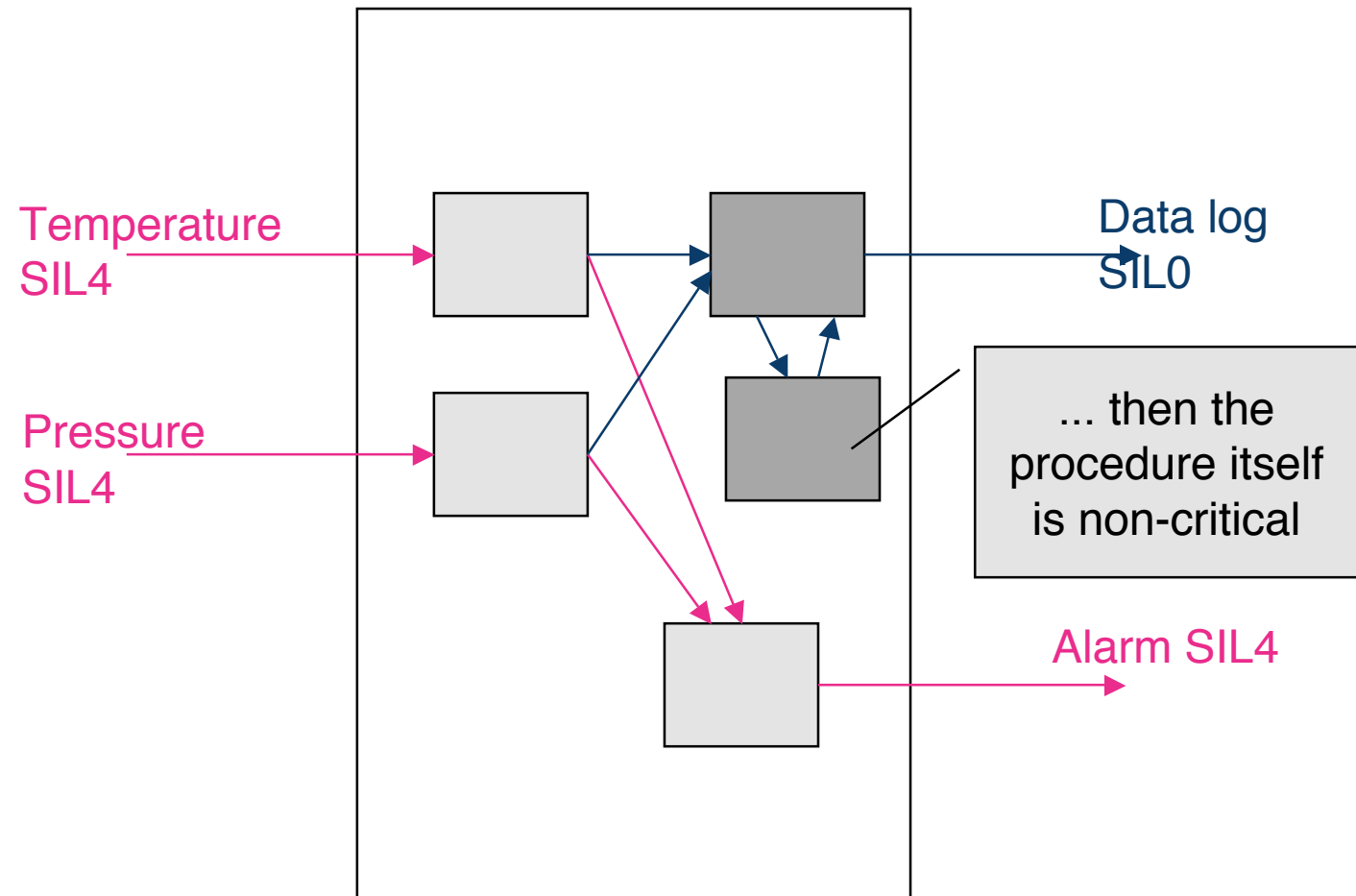


Subprogram level information flow



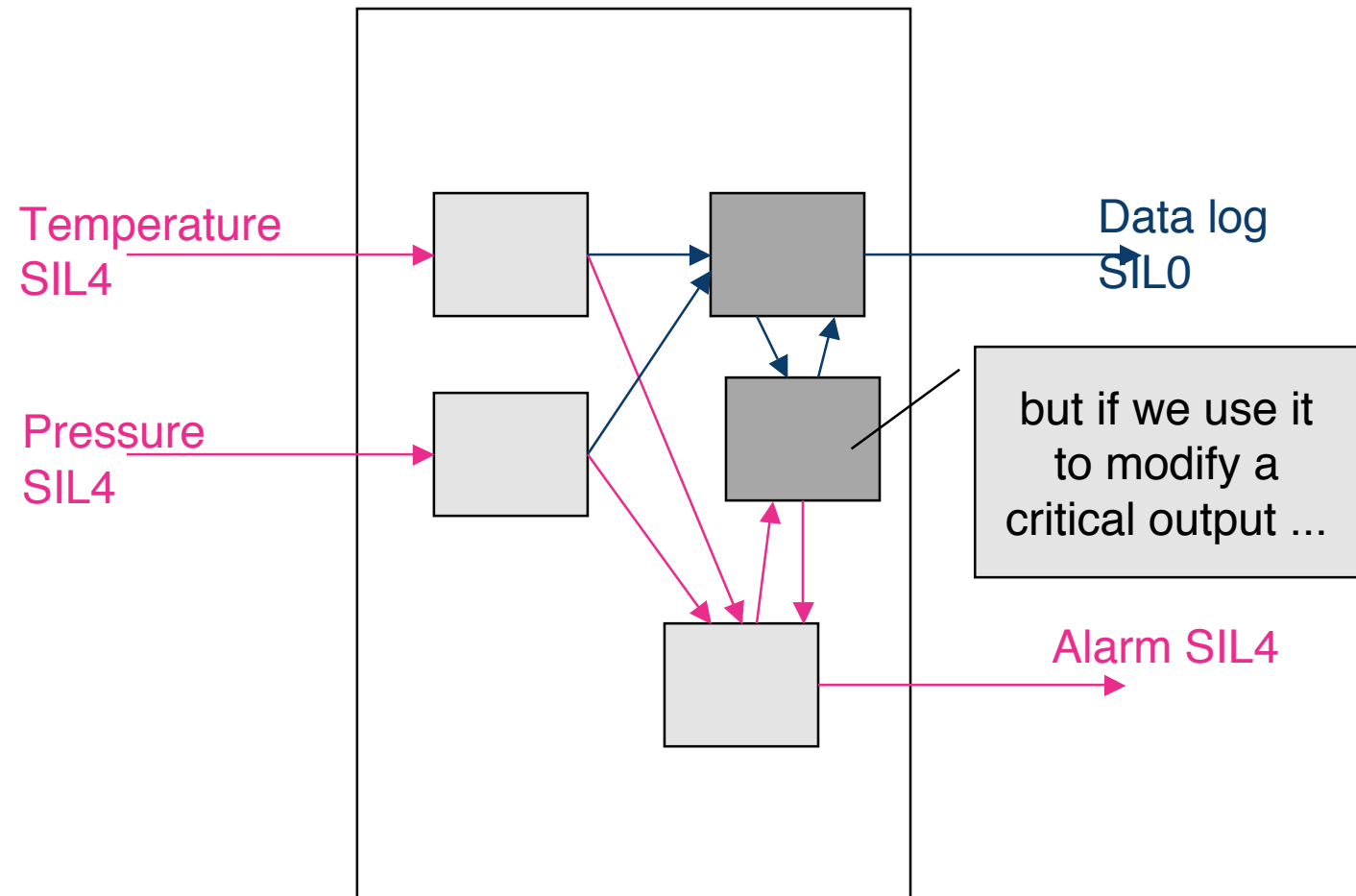


Subprogram level information flow



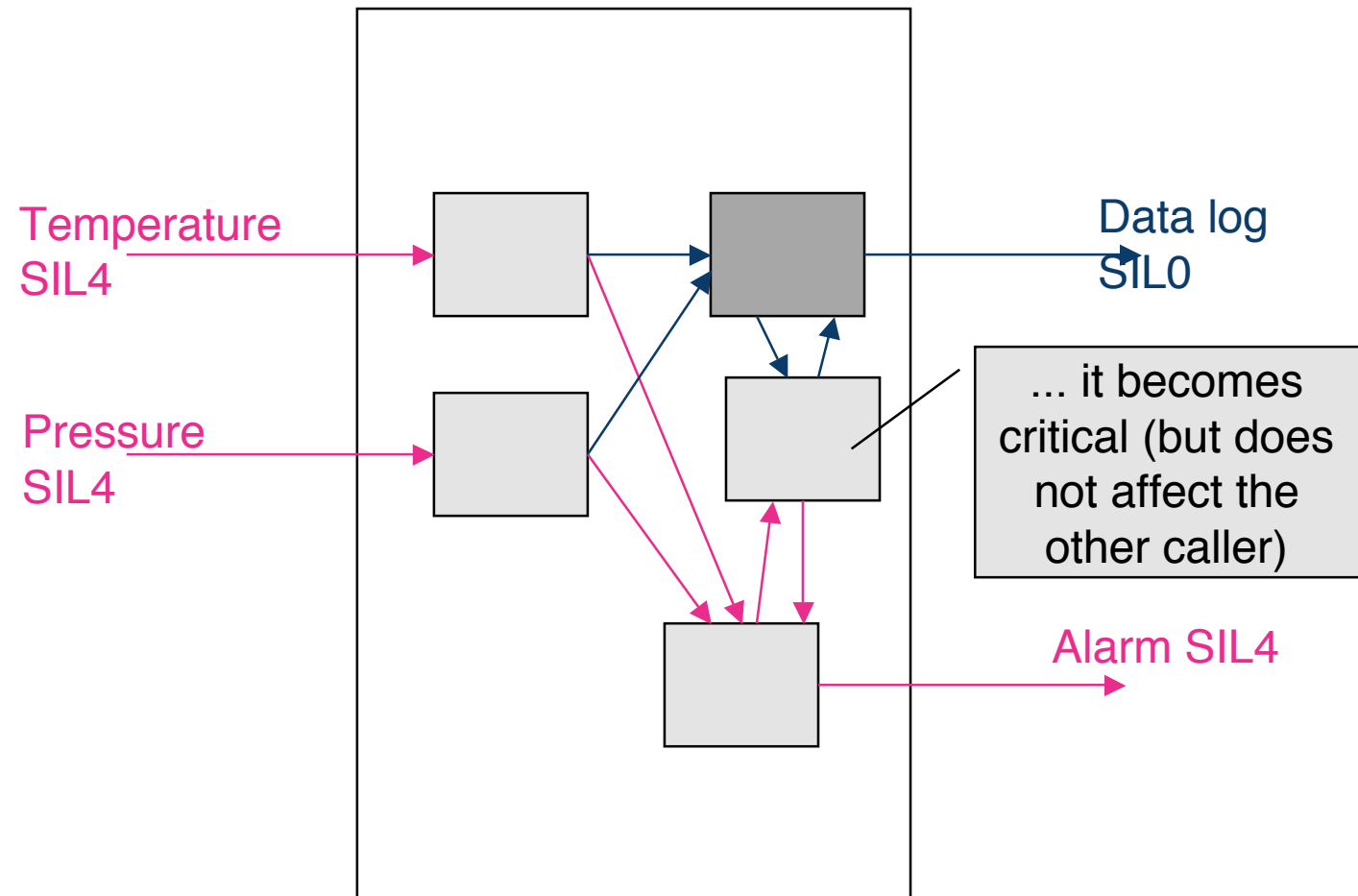


Subprogram level information flow





Subprogram level information flow





Agenda

- The segregation problem
- Preconditions for analysis
- Information flow analysis
- SPARK extensions
- **Example**
- Conclusions



Example - Sensors and actuators

```
--# inherit IntegrityLevels;
package Temperature
--# own in Inputs
--#   (Integrity => IntegrityLevels.SIL4);
is
  procedure Read (TheTemperature : out Integer);
  --# global in Inputs;
  --# derives TheTemperature from Inputs;
  --# declare Integrity => IntegrityLevels.SIL4;
end Temperature;

--# inherit IntegrityLevels;
package DataLog
--# own out Outputs
--#   (Integrity => IntegrityLevels.SIL1);
is
  procedure PutTemp (TheTemperature : in Integer);
  --# global out Outputs;
  --# derives Outputs from TheTemperature;
  --# declare Integrity => IntegrityLevels.SIL1;
end DataLog;
```



Example

```
--# inherit IntegrityLevels;
package Temperature
--# own in Inputs
--#   (Integrity => IntegrityLevels.SIL4);
is
  procedure Read (TheTemperature : out Integer);
  --# global in Inputs;
  --# derives TheTemperature from Inputs;
  --# declare Integrity => IntegrityLevels.SIL4;
end Temperature;

--# inherit IntegrityLevels;
package DataLog
--# own out Outputs
--#   (Integrity => IntegrityLevels.SIL1);
is
  procedure PutTemp (TheTemperature : in Integer);
  --# global out Outputs;
  --# derives Outputs from TheTemperature;
  --# declare Integrity => IntegrityLevels.SIL1;
end DataLog;
```



Example

```
--# inherit IntegrityLevels;
package Temperature
--# own in Inputs
--#   (Integrity => IntegrityLevels)
is
  procedure Read (TheTemperature : in Integer);
  --# global in Inputs;
  --# derives TheTemperature from Inputs;
  --# declare Integrity => IntegrityLevels;
end Temperature;

--# inherit IntegrityLevels;
package DataLog
--# own out Outputs
--#   (Integrity => IntegrityLevels.SIL1);
is
  procedure PutTemp (TheTemperature : in Integer);
  --# global out Outputs;
  --# derives Outputs from TheTemperature;
  --# declare Integrity => IntegrityLevels.SIL1;
end DataLog;
```

```
package IntegrityLevels
is
  SIL4 : constant := 4;
  SIL3 : constant := 3;
  SIL2 : constant := 2;
  SIL1 : constant := 1;
  SIL0 : constant := 0;
end IntegrityLevels;
```



Example - new annotation

```
--# inherit IntegrityLevels;
package Temperature
--# own in Inputs
--#   (Integrity => IntegrityLevels.SIL4);
is
  procedure Read (TheTemperature : out Integer);
  --# global in Inputs;
  --# derives TheTemperature from Inputs;
  --# declare Integrity => IntegrityLevels.SIL4;
end Temperature;

--# inherit IntegrityLevels;
package DataLog
--# own out Outputs
--#   (Integrity => IntegrityLevels.SIL1);
is
  procedure PutTemp (TheTemperature : in Integer);
  --# global out Outputs;
  --# derives Outputs from TheTemperature;
  --# declare Integrity => IntegrityLevels.SIL1;
end DataLog;
```



Example - Main control loop

```
with Temperature, DataLog;
--# inherit Temperature, DataLog;
--# main_program;
procedure Main
--# global in      Temperature.Inputs;
--#               out DataLog.Outputs;
--# derives DataLog.Outputs from
--#               Temperature.Inputs;
--# declare Integrity => IntegrityLevels.SIL4;
is
    CurrentTemperature : Integer;
begin
    loop
        Temperature.Read (CurrentTemperature);
        DataLog.PutTemp (CurrentTemperature);
        -- other activities not shown
    end loop;
end Main;
```



Example - Main control loop

```
with Temperature, DataLog;
--# inherit Temperature, DataLog;
--# main_program;
procedure Main
--# global in      Temperature.Inputs;
--#               out DataLog.Outputs;
--# derives DataLog.Outputs from
--#               Temperature.Inputs;
--# declare Integrity => IntegrityLevels.SIL4;
is
    CurrentTemperature : Integer;
begin
    loop
        Temperature.Read (CurrentTemperature);
        DataLog.PutTemp (CurrentTemperature);
        -- other activities not shown
    end loop;
end Main;
```

Everything is fine: non-critical output is being derived from a trusted, critical input. Each subprogram is handling the right level of data.



Example - Alarm package

```
--# inherit IntegrityLevels;  
package Alarm  
--# own out Outputs  
--#   (Integrity => IntegrityLevels.SIL4);  
is  
    procedure Sound;  
    --# global out Outputs;  
    --# derives Outputs from ;  
    --# declare Integrity => IntegrityLevels.SIL4;  
end Alarm;
```



Example - Alarm package

```
--# inherit IntegrityLevels;  
package Alarm  
--# own out Outputs  
--#   (Integrity => IntegrityLevels.SIL4);  
is  
  procedure Sound;  
  --# global out Outputs;  
  --# derives Outputs from ;  
  --# declare Integrity => IntegrityLevels.SIL4;  
end Alarm;
```

Design decision: from where should we call the Alarm package?



Example - from the data logger?

```
--# inherit IntegrityLevels;
package DataLog
--# own out Outputs
--#   (Integrity => IntegrityLevels.SIL1);
is
    procedure PutTemp (TheTemperature : in Integer);
--# global out Outputs;
--# derives Outputs from TheTemperature;
--# declare Integrity => IntegrityLevels.SIL1;
end DataLog;
```



Example - from the data logger?

```
--# inherit IntegrityLevels, Alarm;
package DataLog
--# own out Outputs
--#   (Integrity => IntegrityLevels.SIL1);
is
  procedure PutTemp (TheTemperature : in Integer);
  --# global out Outputs;
  --#   out Alarm.Outputs;
  --# derives Outputs
  --#   Alarm.Outputs from TheTemperature;
  --# declare Integrity => IntegrityLevels.SIL1;
end DataLog;
```



Example - from the data logger?

```
--# inherit IntegrityLevels, Alarm;
package DataLog
--# own out Outputs
--#   (Integrity => IntegrityLevels.SIL1);
is
  procedure PutTemp (TheTemperature : in Integer);
  --# global out Outputs;
  --#   out Alarm.Outputs;
  --# derives Outputs
  --#   Alarm.Outputs from TheTemperature;
  --# declare Integrity => IntegrityLevels.SIL1;
end DataLog;
```

SIL4 output being
modified by a SIL1
procedure!



Example - in the main control loop?

```
with Temperature, DataLog;
--# inherit Temperature, DataLog;
--# main_program;
procedure Main
--# global in      Temperature.Inputs;
--#               out DataLog.Outputs;
--# derives DataLog.Outputs from
--#               Temperature.Inputs;
--# declare Integrity => IntegrityLevels.SIL4;
is
    CurrentTemperature : Integer;
begin
    loop
        Temperature.Read (CurrentTemperature);
        DataLog.PutTemp (CurrentTemperature);
    end loop;
end Main;
```



Example - in the main control loop?

```
with Temperature, DataLog, Alarm;
--# inherit Temperature, DataLog, Alarm;
--# main_program;
procedure Main
--# global in      Temperature.Inputs;
--#              out DataLog.Outputs;
--# derives DataLog.Outputs
--#           Alarm.Outputs from
--#           Temperature.Inputs;
--# declare Integrity => IntegrityLevels.SIL4;
is
    CurrentTemperature : Integer;
    MaxTemp : constant Integer := 100;
begin
    loop
        Temperature.Read (CurrentTemperature);
        DataLog.PutTemp (CurrentTemperature);
        if CurrentTemperature > MaxTemp then
            Alarm.Sound;
        end if;
    end loop;
end Main;
```



Agenda

- The segregation problem
- Preconditions for analysis
- Information flow analysis
- SPARK extensions
- Example
- **Conclusions**



Conclusions

- A mathematically precise proof of non-interference is feasible ...
 - ... but only in the sound environment provided by SPARK
- The approach allows verification effort to be focussed:
 - reducing cost
 - and, potentially, raising quality
- Has been done manually on a 00-55 SIL4 project and accepted by ISA
- Objective-based standards will make such arguments more acceptable in