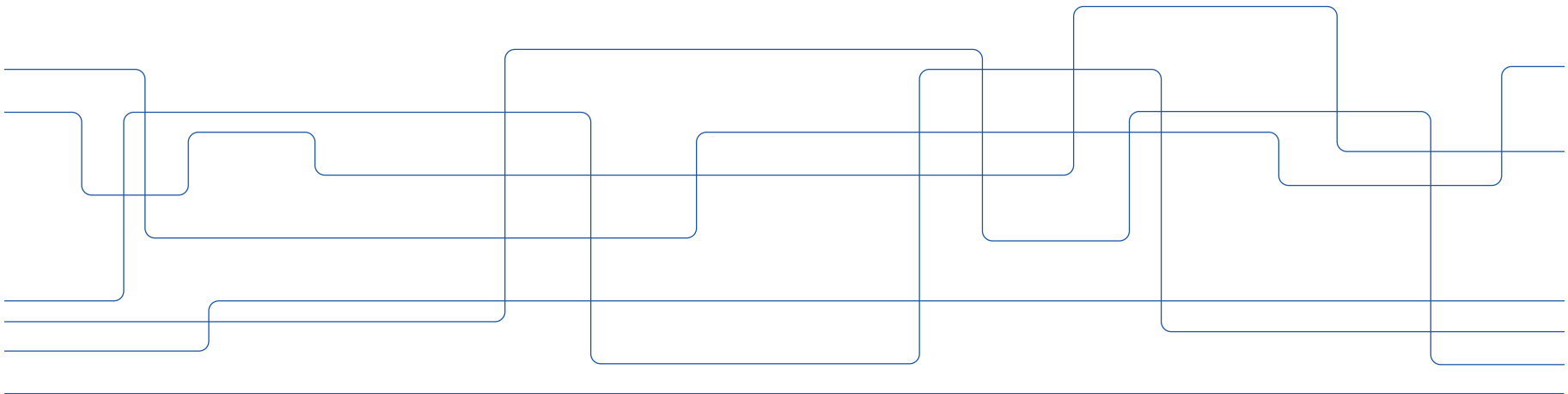


# DD2460.Lecture 6

Refinement in Event-B. Safety case.





# Refinement

- Refinement is a process that is used describe any or all of the following changes to a model:
  - extended functionality: we add more functionality to the model, perhaps modelling the requirements for a system in layers;
  - more detail: we give a finer-grained model of the events. This is often described as moving from the abstract to the concrete (from “what” to “how”)
  - changing state model: we change the way that the state is modelled, but also describe how the new state models the old state (data refinement)
-



# Correct-by-construction development: formal meaning

- In all cases of refinement, the behaviour of the refined machine must be consistent with the behaviour of the machine being refined (more abstract machine).
- Observe consistent does not mean equivalent:
  - the behaviour of the refined machine does not have to be the same, but the behaviour must not contradict the behaviour of the machine being refined.
  - e.g., machines may be nondeterministic and the refined machine may remove some of the nondeterminism.

# Refinement machine: refined state (1/2)

- The refinement machine consists of refined state and refined events
- A refined state is logically a new state.
- The refined state must contain a refinement relation that expresses how the refined state models the state being refined.
- The refined state may contain variables that are syntactically and semantically equivalent to variables in the state of the machine being refined.
- In that case, the new and old variables are implicitly related by an equivalence relation.



# Refinement machine: refined and new events

## (2/2)

- Refined events logically refine the events of the refined machine.
- The refined events are considered to simulate the behaviour of the events being refined, where the effects of the refined events are interpreted through the refinement relation.
- New events add new functionality to the model.
- The new events must not add behaviour that is inconsistent with the behaviour of the refined machine.

# Refinement relation

- The refinement relation is expressed explicitly or implicitly in the invariant of a refinement
  - It relates the state of the machine being refined to the state of the refinement machine.
- Refinement consistency means that any behaviour of a refined event must be acceptable behaviour of the unrefined event in the unrefined model.
- An informal example: if in a restaurant you asked for fish or vegetables as the main course and you are given fish then it is consistent with your request (a valid refinement). But if you are given meat then it is not acceptable, i.e., not a valid refinement

# Rules of refinement

- In the refinement we can:
- **strengthen guards and invariants:**
  - guards and invariants can be strengthened, provided overall functionality is not reduced (no new deadlocks are introduced);
- **nondeterminism can be reduced:**
  - where a model offers choice, then the choice can be reduced (but not increased) in the refinement;
- **the state may be augmented by an orthogonal state:**
  - new state variables, whose values do not affect the existing state, may be added.

# Once again about events in refinement

- What might happen during refinement:
  - a single event may be refined by multiple events, or
  - vice versa multiple events may be refined by a single event.
  - Refinement may introduce new events.
- Important: The new events must not change variables inherited from the state of the refined machine.
- This is a restriction that recognises that a machine state can be modified only by the events of that machine, or their refinements.
- Our informal restaurant example: in the refinement you can add one or several other courses, i.e. starter or/and desert but you cannot change the alternatives that you had for the main course



# Example of refinement: coffee club

- We start by specifying functionality of a simple piggybank system for collecting and spending money for coffee.

Requirements document:

**REQ1:** a money bank for storing and reclaiming finite, non-negative funds for a coffee club;

**REQ2:** an operation for adding money to the money bank;

**REQ3:** an operation for removing money from the money bank; cannot remove more than money bank





# CoffeeClub Abstract Specification (1/2)

**MACHINE** CoffeeClub

**VARIABLES** *piggybank* // Denotes money bank for coffee club

**INVARIANTS**

*inv1*:  $\text{piggybank} \in \text{NAT}$  // REG1: piggybank should be non-negative

**EVENTS**

**INITIALISATION**  $\triangleq$

then

*act1*:  $\text{piggybank} := 0$  // But could also initialize to any natural number

end

# CoffeeClub Abstract Specification (2/2)

```
FeedBank  $\triangleq$                                      // REQ2: adding money to piggybank.  
  any amount  
  where  
    grd1: amount  $\in \text{NAT1}$   
  then  
    act1: piggybank := piggybank+amount  
  end  
RobBank  $\triangleq$                                        // REQ3: removing money from piggybank.  
  any amount  
  where  
    grd1: amount  $\in 1..\text{piggybank}$   
  then  
    act1: piggybank := piggybank-amount  
  end
```



# Proof obligations

Sequent representation:

*hypothesis*  $\vdash$  *goal*

Proof obligations are the checks showing that the specification is consistent with formal constraints of the model

*hypothesis*  $\vdash$  *goal* means that *the truth of the hypotheses leads to the truth of the goal*.

Note:

1. If any of the hypotheses is false ( $\perp$ ) then any goal is trivially established.
2. If the hypotheses are identically true ( $\top$ ) then the hypotheses will be omitted.

# Discharging POs

- Important to understand that the consequent should be provable from the given hypotheses;
  - there is nothing else in the form of a hypothesis that should be required.
- If the PO cannot be discharged then there are many cases that must be considered, of which
- the invariants are too strong/weak
- the guards are too weak/strong;
- the actions are inappropriate/incomplete



# Discharging POs not the goal in itself

- Working with Pos is not primarily about discharging the proof obligations, it's about determining whether the model is consistent with the requirements and internally consistent.
- Observe that the proof obligations might be discharged, but the model may not be what is required.



# Refinement of the CoffeeClub

- Abstract specification of CoffeeClub is very simple:
- *piggybank* models an amount of money
- Events *FeedBank* and *RobBank* describe adding to or taking from amount modelled by piggybank.
- We will now model behaviour that describes club-like behaviour for members who want to be able to purchase cups of coffee.



# Additional requirements for coffee club

- The new requirements are:

**REQ4:** a facility for members to join the coffee club; each member has a distinct membership id;

**REQ5:** members have an account that cannot go into debt;

**REQ6:** an operation that enables a member to add money to their account;

**REQ7:** money added to a members account is also added to the club money bank;

**REQ8:** an operation that sets the price for a cup of coffee;

**REQ9:** an operation that enables a member to buy a cup of coffee; the member's account is reduced by the cost of a cup of coffee;



# Refinement: new variables and events

- We will introduce variables *members*, *accounts* and *coffeeprice*
- New events that correspond to
- *a new member joining the club*: each member of the club is represented by a unique identifier that is arbitrarily chosen from an abstract set *MEMBERS*;
- *a member adding money to their account*: each member has an account, to which they can add “money”;
- *a member buying a cup of coffee*: there will be a variable, *coffeeprice*, representing the cost of a cup of coffee, and each member can buy a cup of coffee provided they have enough money in their account.
- *The value of all money added to accounts is added to piggybank* (connection to abstract state space)



# Refinement: defining context

CONTEXT MembersContext

SETS

MEMBER REQ4

AXIOMS

axm1:  $\text{finite}(\text{MEMBER})$

END



# Refinement: defining new variables

**MACHINE** MemberShip

REFINES

CoffeeClub

SEES

Members

VARIABLES

piggybank

members     REQ4: the set of current members

accounts     REQ5: the member accounts

coffeeprice     REQ8: the price of a cup of coffee

INVARIANTS

inv1:      $\text{piggybank} \in \mathbb{N}$

inv2:      $\text{members} \subseteq \text{MEMBER}$      REQ4: each member has unique id

inv3:      $\text{accounts} \in \text{members} \rightarrow \mathbb{N}$      REQ5: each member has an account

inv4:      $\text{coffeeprice} \in \mathbb{N1}$      REQ8: any price other than free!

# Refinement: initialisation

## EVENTS

**Initialisation** : *extended*  $\hat{=}$

## THEN

act2:    *members* :=  $\emptyset$       empty set of members

act3:    *accounts* :=  $\emptyset$       empty set of accounts

act4:    *coffeeprice* :  $\in \mathbb{N}1$     initial coffee price set to arbitrary non-zero value

## END

- In extended mode, only the new parameters, guards and actions are displayed, that is, only the parts of an event that extend the event being refined.



# Refinement: new events for setting price and adding member

**SetPrice**  $\hat{=}$  REQ8

ANY

amount

WHERE

grd1:  $amount \in \mathbb{N}_1$

THEN

act1:  $coffeeprice := amount$

END

**NewMember**  $\hat{=}$  REQ4

ANY

member

WHERE

grd1:  $member \in MEMBER \setminus members$

THEN

act1:  $members := members \cup \{member\}$

act2:  $accounts(member) := 0$

END



# Refinement: new events for adding money and buying coffee

**Contribute**  $\hat{=}$  REQ6

ANY

amount

member

WHERE

grd1:  $amount \in \mathbb{N}_1$

grd2:  $member \in members$

THEN

act1:  $accounts(member) := accounts(member) + amount$

act2:  $piggybank := piggybank + amount$  REQ7

END

**BuyCoffee**  $\hat{=}$  REQ9

ANY

member

WHERE

grd1:  $member \in members$

grd2:  $accounts(member) \geq coffeeprice$

THEN

act1:  $accounts(member) := accounts(member) - coffeeprice$

END



# Refinement: “old events”

“Old” events remain unchanged. In the extended mode they are “hidden”

**FeedBank** : *extended*  $\hat{=}$

REFINES

FeedBank

ANY

WHERE

THEN

END

**RobBank** : *extended*  $\hat{=}$

REFINES

RobBank

ANY

WHERE

THEN

END



# Unproved PO: why?

Contribute/piggybank/EQL:

$$\begin{array}{l} \text{amount} \in \mathbb{N}1 \\ \text{member} \in \text{members} \end{array} \vdash \text{piggybank} = \text{piggybank} + \text{amount}$$

- Why cannot we prove it?



# Unproved PO: why?

Contribute/piggybank/EQL:

$$\frac{amount \in \mathbb{N}1}{member \in members} \vdash piggybank = piggybank + amount$$

- Why cannot we prove it?
- EQL PO requires a proof that *piggybank* is not changed, but of course, *piggybank := piggybank + amount* must change the value of the variable *piggybank*, unless *amount* is 0.
- *Contribute* appears in the refinement as a new event, but here it is changing the value of the variable *piggybank*, which is part of the state of *CoffeeClub*, the machine being refined.
- To preserve consistency, any event of a refinement that modifies the state of the machine being refined must itself be a refinement of one or more events of the machine being refined.

# Corrected event:

**Contribute**  $\hat{=}$

REFINES

FeedBank

ANY

amount

member

WHERE

grd1:  $amount \in \mathbb{N}_1$

grd2:  $member \in members$

THEN

act1:  $piggybank := piggybank + amount$

act2:  $accounts(member) := accounts(member) + amount$

END

- The event *FeedBank* of *CoffeeClub* changes the value of the variable *piggybank* in a similar way to *Contribute*, thus *Contribute* must be seen as a refinement of *FeedBank*



# Lesson learnt

- Usually the presence of undischarged EQL POs will probably indicate a bad refinement.
- Check that your working with the “old” variables is consistent with your abstract specification.

# Types of POs

- You are all familiar with INV type of POs: proving that invariant is preserved by the initialisation and events
- Now we have learnt about EQL POs: demonstrating consistency of refinement wrt more abstract specification
- WD: *well-defined* Some expressions, especially function applications, may not be defined everywhere. For example,  $f(x)$  is only defined if  $x$  is in the domain of  $f$ , ie  $x \in \text{dom}(f)$ .
- FIS: *feasibility*. Specifying a property with a predicate does not carry with it the promise that there exist solutions that satisfy the predicate.
- e.g.  $x + 1 = x - 1$  cannot be satisfied by any  $x \in \mathbb{N}$ . Feasibility required to show that instances that satisfy a predicate do exist.



# Safety case

Safety case presents the argument that a system will be acceptably safe in a given context

- Many standards establish the need for production of a safety case, e.g.
  - *“Safety Cases are required for all new ships and equipment as a means of formally documenting the adequate control of Risk and demonstrating that levels of risk achieved are As Low As Reasonably Practicable (ALARP).” (JSP430 Ship Safety Management regulations)*
  - *A person in control of any railway infrastructure shall not use or permit it to be used for the operation of trains unless*
    - (a) he has prepared a safety case ...*
    - (b) the Executive has accepted that safety case ...”*
  - *(HSE Railway Safety Case Regulations)*
- *“The Software Design Authority shall provide a Software Safety Case ...” (U.K. Defence Standard 00-55)*



# Some definitions

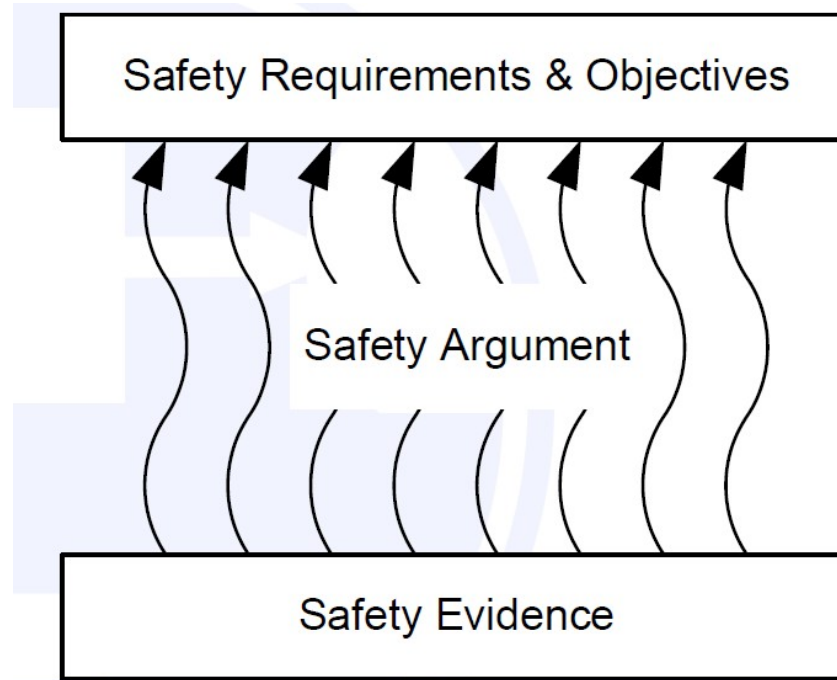
- "A safety case is a comprehensive and structured set of safety documentation which is aimed to ensure that the safety of a specific vessel or equipment can be demonstrated by reference to:
    - safety arrangements and organisation
    - safety analyses
    - compliance with the standards and best practice
    - acceptance tests
    - audits
    - inspections
    - Feedback
    - Another definition
  - ***"The software safety case shall present a well-organised and reasoned justification based on objective evidence, that the software does or will satisfy the safety aspects of the Statement of Technical Requirements and the Software Requirements specification."***
-



# Argument and evidence

- **A safety case requires two elements:**
    - ***Supporting Evidence***
      - Results of observing, analysing, testing, simulating and estimating the properties of a system that provide the *fundamental* information from which safety can be inferred
    - ***High Level Argument***
      - Explanation of how the available evidence can be reasonably interpreted as indicating acceptable safety – usually by demonstrating compliance with requirements, sufficient mitigation / avoidance of hazards etc
  - **Argument without Evidence is unfounded**
  - **Evidence without Argument is unexplained**
-


# Argument and evidence



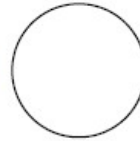


# Goal structuring notation (GSN)

## Purpose of a Goal Structure

To show how **goals**  are broken down into sub-goals,

and eventually supported by evidence (**solutions**)

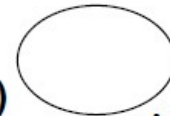


whilst making clear the **strategies**



adopted,

the rationale for the approach (**assumptions, justifications**)



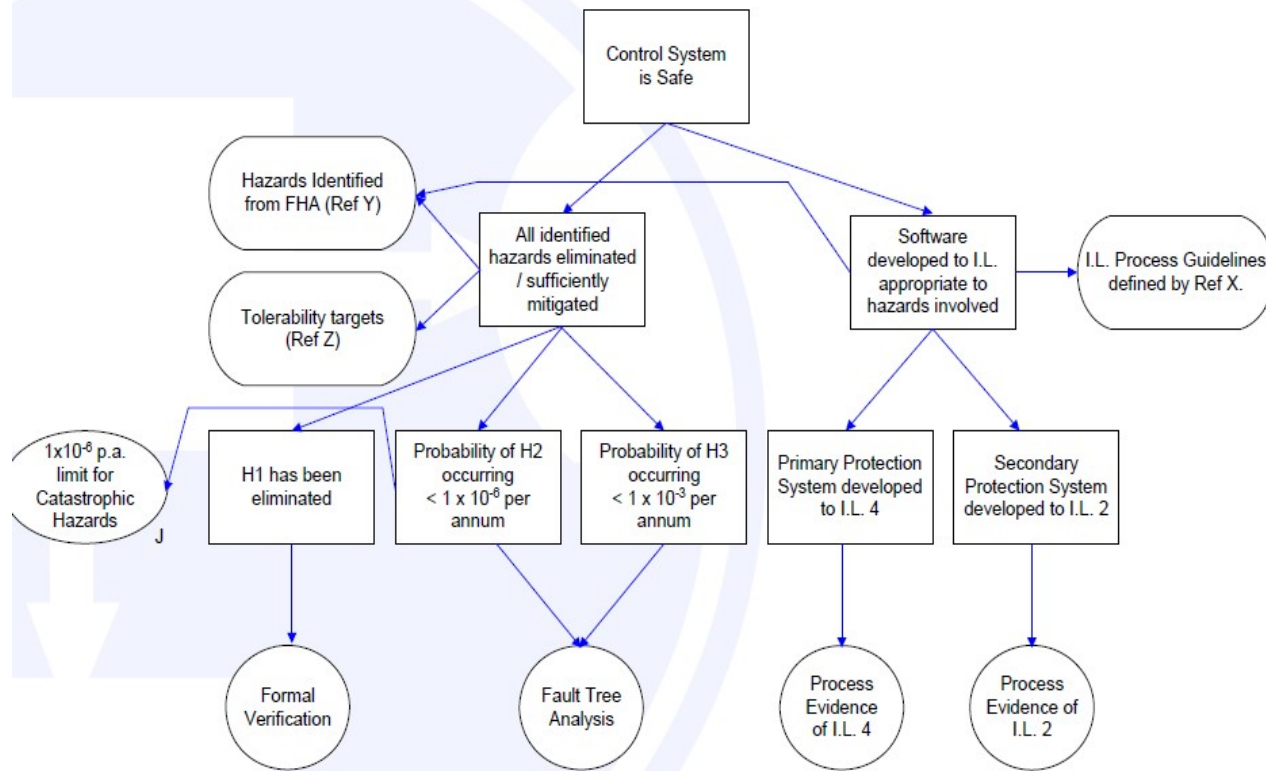
A/J

and the **context**

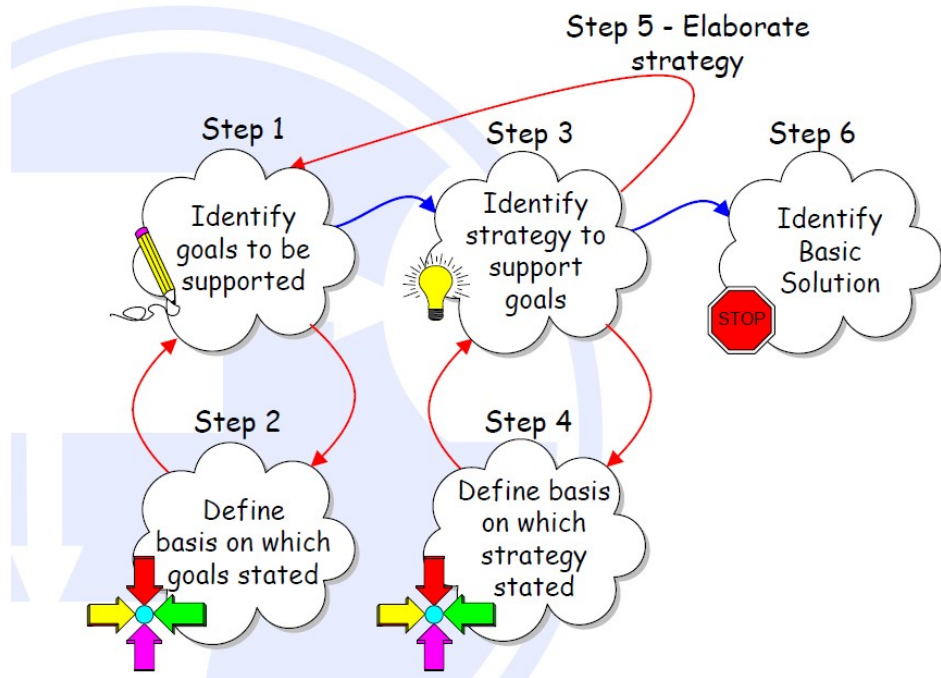


in which goals are stated

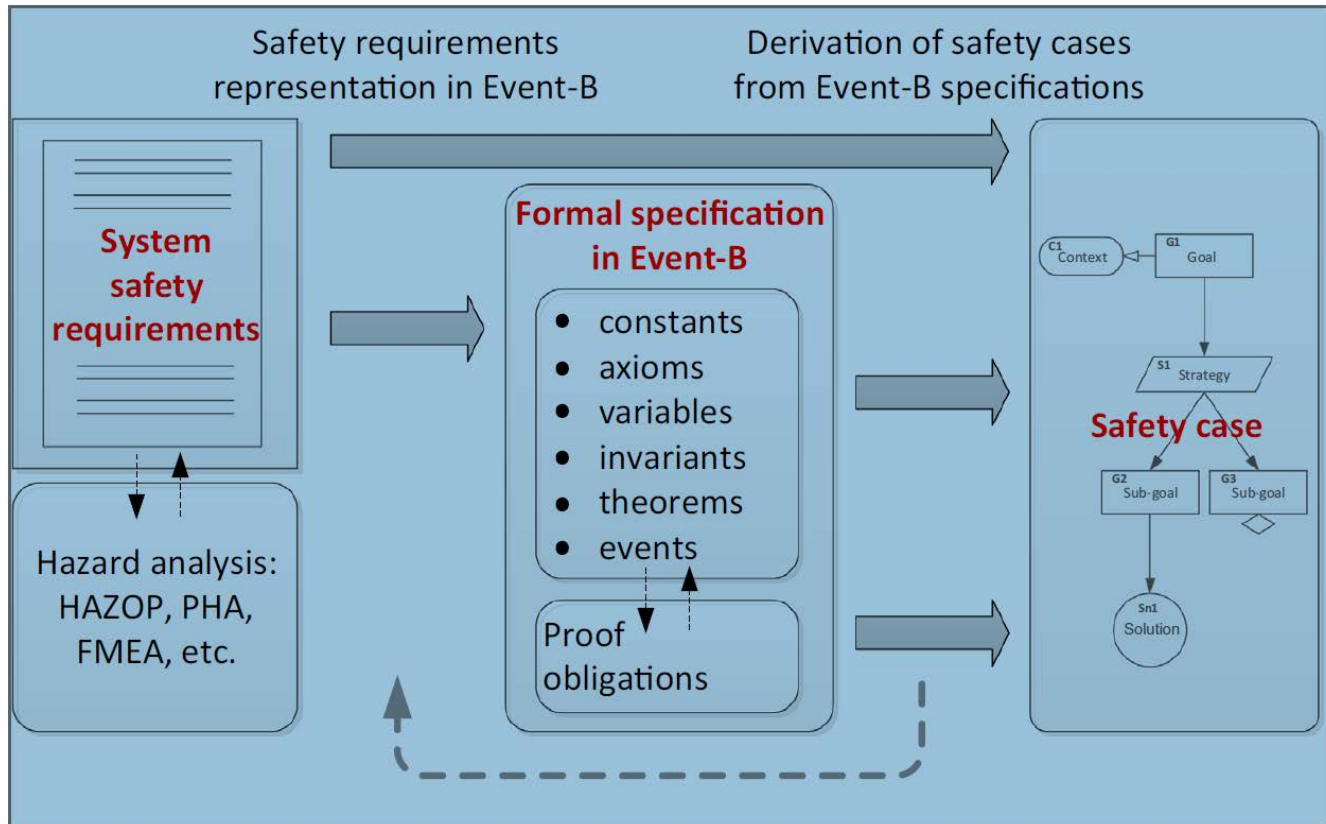
# A simple goal structure



# Strategy



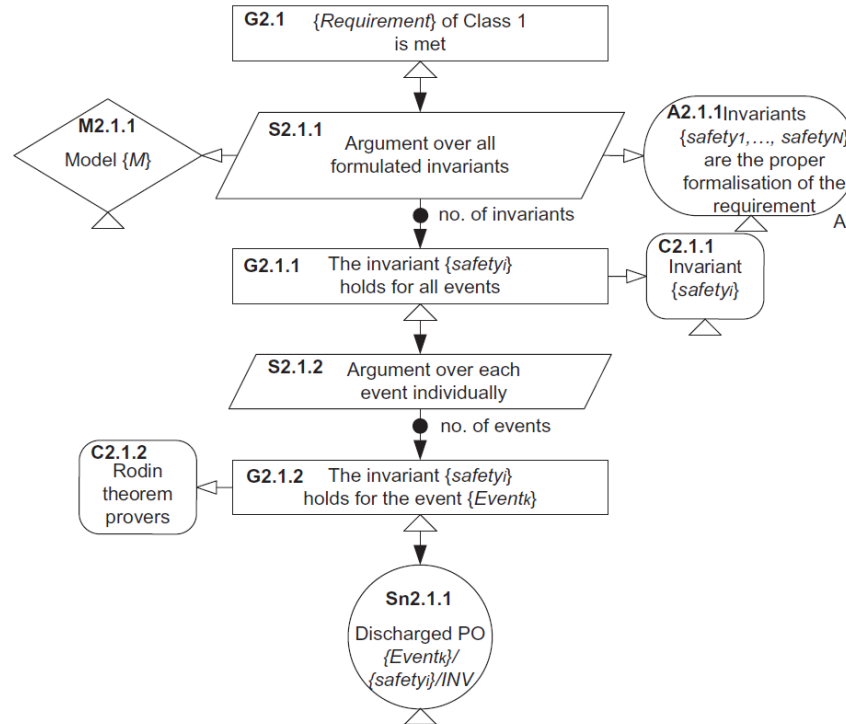
# Constructing safety case based on Event-B specification



# Modelling safety requirements

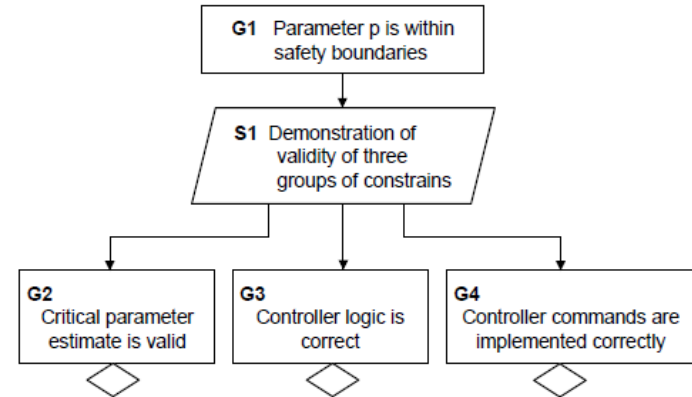
- Several types of safety requirements (SRs) that we often model:
  - SRs about global properties
  - SRs about local properties (at specific state of the system)
  - SRs about control flow (certain order of events)
  - SRs about system termination (e.g. shut down)
- We model these requirements as invariants or via guards or specific actions.
  - By discharging the corresponding proof obligations we produce the evidence that the SR has been met.

# Pattern for safety case fragment about global safety property

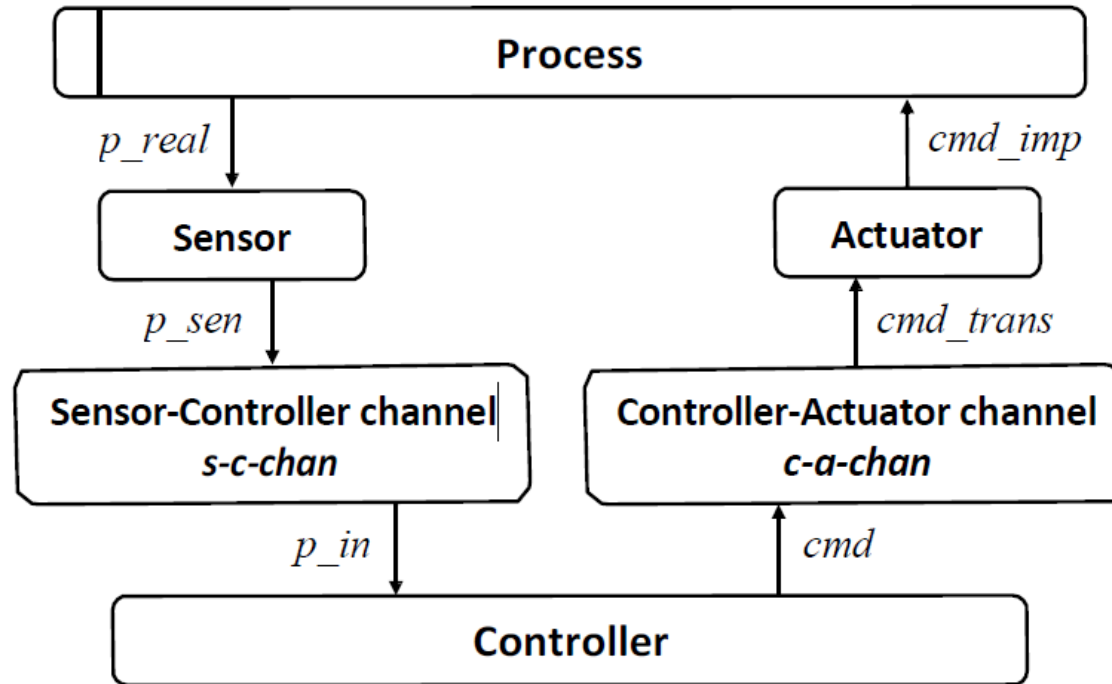


# Safety-critical control system: high-level safety case

- A safety-critical control systems has three “main” components
- Sensing, control and actuation
- If safety is defined in terms of some parameter being within certain safe boundaries then a high level safety case has the following structure:



# Safety-critical networked control system: data flow view

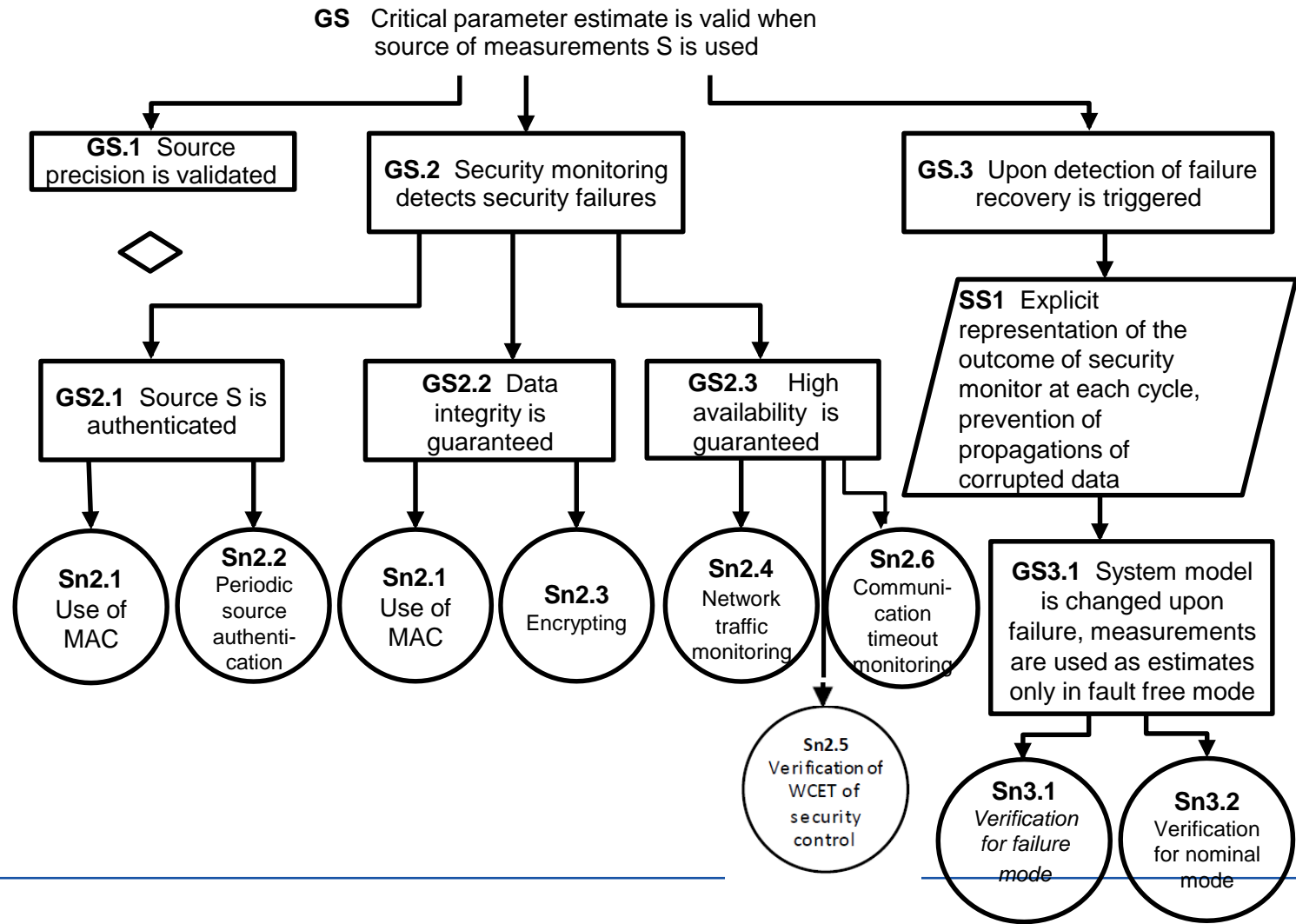






# Safety-security requirements interactions

- Next slide: a fragment of safety case about sensing
- It is important to recognise that undetected failures or cyberattacks can jeopardise safety





# Event-B module wrap-up

- We have studied how to model control systems in Event-B and reason about their safety properties
- We have learnt how to use functions and relations to model various access control functions
- It allows us to demonstrate that no unauthorised access to some resources is possible
- Finally, we have learnt to use proofs as the “debugging” mechanism
- We have learnt to write a structured requirements document and create a safety case
- Modelling dynamic properties such as liveness is not straightforward in Event-B
- So welcome a new topic – Model checking!