DD2460 Lecture 3. Introduction to formal specification

Elena Troubitsyna

About me

- I am associated professor at Theoretical Computer Science division, EECS school
- My research interests focus on formal modelling and verification of dependable systems
- Dependable means trustworthy, i.e., safe, reliable, secure, fault tolerant etc.
- I work mostly on formal specification methods and try to augment them with the capabilities to specify, reason and assess various dependability attributes.
- In this course, I am responsible for Event-B module.
- We will focus on specification and refinement-based development of safety-critical systems and representing the impact of security attacks on safety
- We will work with Rodin platform – a tool for specification and verification in Event-B

Lecture outline

- Why formal specification?
- Safety-critical control systems: structure
- What is safety and how to express it?
- Failures and their impact on safety

Video

- Please watch 9 minutes of video by one of the pioneers of formal methods Prof. Eric Hehner (University of Toronto, Canada):

- https://www.youtube.com/watch?v=89fKiaMxHrA

Questions for the discussion:

- How a program is considered by formal methods?
- Theory is a combination of formalism and rules of proof, calculation, manipulation. What does theory give to a software developer?
- What is a difference between testing and proof-based verification?
- What is the main idea behind correct-by-construction development?

What is a formal specification?

- A **formal specification** is the expression, in some **formal** language and at the some level of abstraction, of a collection of **properties** some **system** should satisfy.
- The formal specification depends on
  - what does "*system*" mean, i.e., where one draws the boundaries,
  - what kind of *properties* are of interest,
  - what level of abstraction is considered, and
  - what kind of *formal language* is used.

The “system” being specified may be:

- a descriptive model of the domain of interest;
- a prescriptive model of the software and its environment;
- a prescriptive model of the software alone;
- a model for the user interface;
- the software architecture;
- a model of some process to be followed;
- etc.

The “properties” under consideration may refer to:

- high-level goals;
- functional requirements;
- non-functional requirements about timing, performance, accuracy, security, etc.;
- environmental assumptions;
- services provided by architectural components;
- protocols of interaction among such components;
- and so on.

Formal specification

- “Formal” is often confused with “precise” (the former entails the latter but the reverse is not true).

- A specification is ***formal*** if it is expressed in a language made of three components:
  - rules for determining the grammatical well-formedness of sentences (the syntax);
  - rules for interpreting sentences in a precise, meaningful way within the considered domain (the semantics);
  - and rules for inferring useful information from the specification (the proof theory).

- The latter component provides the basis for automated analysis of the specification.

Why specify formally?

- Problem specifications are essential for designing, validating, documenting, communicating, reengineering, and reusing solutions.
- Formality helps in obtaining higher-quality specifications within such processes;
    - it also provides the basis for their automated support.
- The act of formalization in itself has been widely experienced to raise many questions and detect serious problems in original informal formulations.
- Besides, the semantics of the formalism being used provides precise rules of interpretation that allow many of the problems with natural language to be overcome. A language with rich structuring facilities may also produce better structured specifications.

Specify... for whom?

- Formal specifications may be of interest to different stakeholders having fairly different background, abstractions and languages:
  - clients
  - domain experts
  - users
  - architects
  - programmers
  - and tools.

Specify... when?

- There are multiple stages in the software life-cycle at which formal specifications may be useful:
  - when modeling the domain;
  - when elaborating the goals, requirements on the software, and assumptions about the environment;
  - when designing a functional model for the software;
  - when designing the software architecture;
  - or when modifying or reengineering the software.

Value of formal specification

- The cost of fixing a specification or design error is higher the later in the development that error is identified.

- ***Boehm's First Law*: *Errors are more frequent during requirements and design activities and are more expensive the later they are removed.***

Specification methods

- Facilitate discovering errors at early stages of system development when they are less expensive to fix.
- Common errors introduced in the early stages of development are errors in understanding the system requirements and errors in writing the system specification.
- Without a rigorous approach to understanding requirements and constructing specifications, it can be very difficult to uncover such errors other than through testing of the software product after a lot of development has already been undertaken.

Why is it difficult?

- Lack of precision in formulating specifications resulting in ambiguities and inconsistencies that are difficult to detect.

- High complexity
  - complexity of requirements;
  - complexity of the operating environment of a system or
  - complexity of the design of a system.

The use of formal modelling

- The main aim is to overcome the problem of lack of precision.
- Formal modelling languages are supported by verification methods that support the discovery and elimination of inconsistencies in models.
- But precision does not address the problem of complex requirements and operating environments.
- Complexity cannot be eliminated but we can try to master it via **abstraction**.

Problem abstraction

- Abstraction can be viewed as a process of simplifying the problem at hand and facilitating our understanding of a system.

- Abstraction should
  - **focus** on the **intended purpose** of the system and
  - **ignore** details of **how** that purpose is achieved.

Abstraction

- If the purpose of the system is to provide some service, then
  - model what a system does from the perspective of the service user.
  - ‘user’ might be computing agents as well as humans

- If the purpose of the system is to control, monitor or protect some phenomena, then
  - the abstraction should focus on those phenomenon, considering in what way they should be monitoring, controlled or protected and should ignore the way in which this is achieved.

System and its boundaries

- A **system** is an entity that interacts with other entities (systems, HW, SW, humans, physical world with its natural phenomena)
- Other entities form the **environment** of the given system

- **System boundary** is a common frontier between the system and its environment
  – Question of boundaries is complex

System function and behavior

- **Function** is what the system is intended to do.
  - Described by functional specification

- **Behaviour** is what the system does to implement its functions
  - Described by a sequence of states

- The **total state** of a system is defined by the conditions of computation, communication, stored information, interconnection, physical conditions etc

System structure

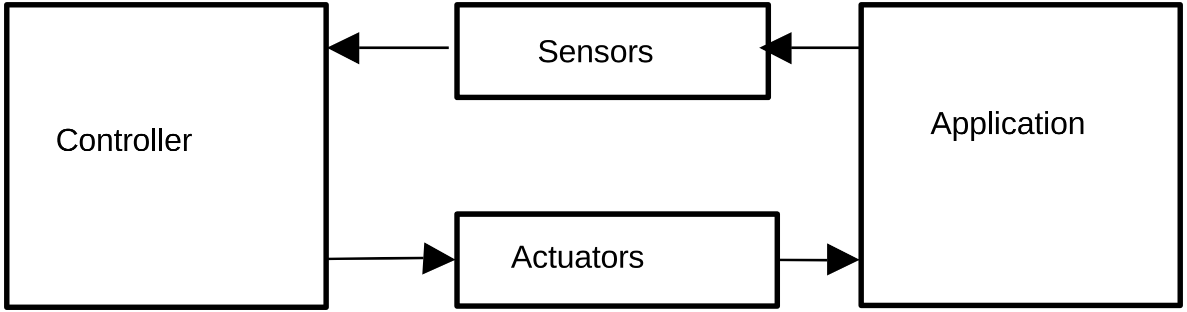
- **Structure** of a system is what enables it to generate the behavior

- It is composed on **components** bound together
  - Each component is another system etc.

- The recursion stops when the component is considered to be **atomic** (cannot be decomposed further or no interest in this)

Generic control system

Safety-critical systems are typically control systems



Generic architecture of a control system

Control system structure

- Main components
- **Application:** A physical entity whose function and operation is being monitored and controlled
- **Controller.** Hardware and software monitoring and controlling the application in real time
- **Actuator (effector).** A device that converts an electrical signal from the output of the computer to a physical quantity, which affects the function of the application.
- **Sensor:** A device that converts an application's physical quantity into an electric signal for input into the computer.

- The behaviour of the system is cyclic. The cycle is called a control loop.
- The control loop is executed once per certain period of time

Control loop

Periodically:

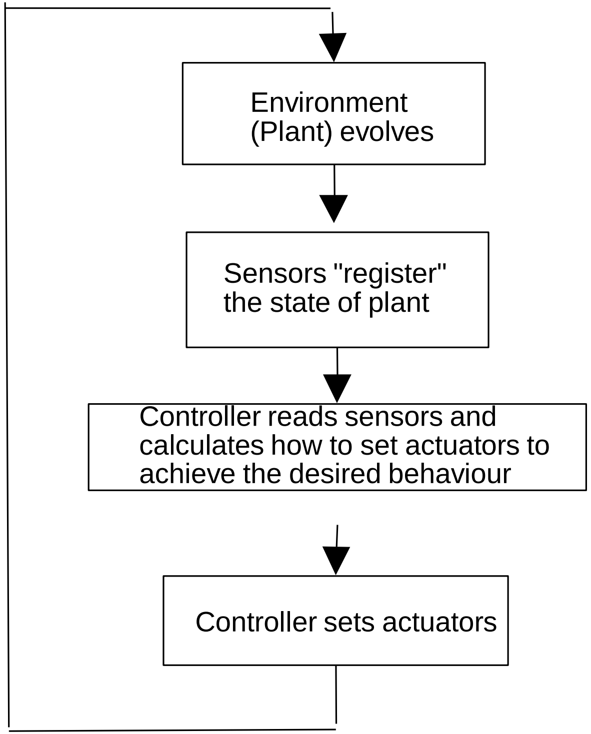
Environment's physical process evolves;

Updating sensors;

Reading sensors;

Computing required control actions;

Setting actuators



Example of a control system: cold vacine storage

- The temperature in a specialized freezer should not exceed minus 70^o Celsius.
- What kind of components the freezer control system should have?

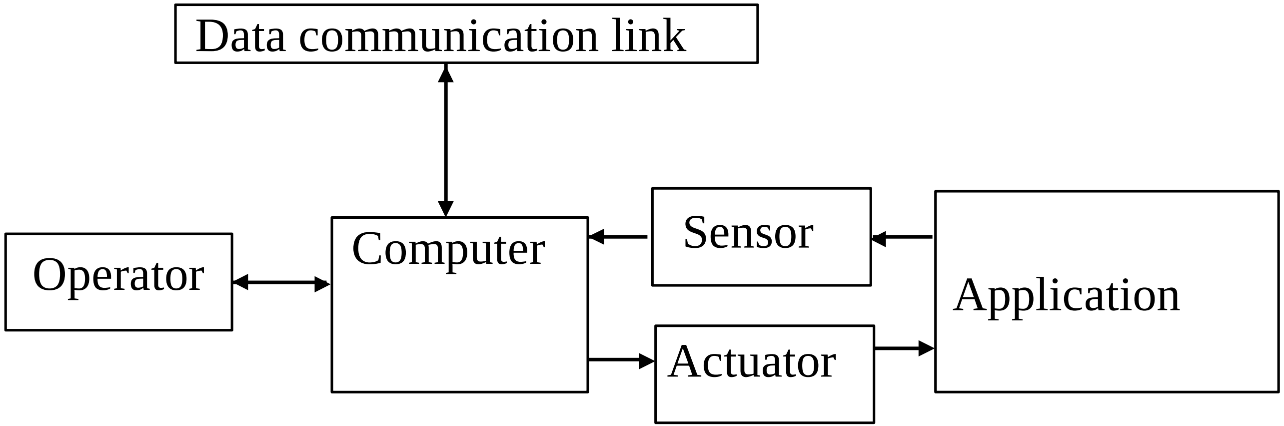
The Pfizer COVID-19 vaccine needs to be stored at minus 70 Celsius. Health care providers will need to store it either in dry ice for shorter stints or in specialized freezers.
Leon Neal/Getty Images

Example of a control system:cold vacine storage

- Application: storage chamber
- Sensor: Temperature sensor
- Actuator: Cooling engine
- Controller:
  - checks measurements
  - sets the cooling engine

  Might also:
  - output information on a display
  - Write to log file and send it over network

A variant of networked control system structure with a human operator



Defining the control cycle for the cold storage control system

- We want to express the following cycling behaviour:
  - Controller receives reading from sensor
  - It decides to increase cooler power if temperature is between -71 and -70 degrees and decrease cooler power if the temperature is between -71 and -72 degrees.
  - If the cooler is in the increased power state then the temperature is decreasing for 0.1 degree per cycle
  - If the cooler is in the decreased power state then the temperature is increasing for 0.1 degree per cycle

Specifying system behaviour (informally)

- The system behaviour is defined in terms of states.
- A state is defined by the values of variables

Variables:

temp: temperature measured by the sensor

cooler: setting of cooler -- increasing or decreasing

phase: variable defining at which phase of the control loop we are: plant, cnt

INIT: phase:= plant; cooler := decr; temp :=70

do (infinitely)

IF phase= plant AND cooler= incr THEN temp := temp -0.1; phase := cnt

IF phase= plant AND cooler= decr THEN temp := temp +0.1; phase := cnt

IF phase = cnt AND $-71 < \text{temp} \leq -70$ then cooler := incr; phase := plant

IF phase = cnt AND $-72 < \text{temp} \leq 71$ then cooler := decr; phase := plant

enddo

Safety

- How do you define safety for the vacine storage system?
- What kind of assumptions do you make?

Safety

- General definition of safety:
- Safety is a property of the system to not cause harm to its users and environment,
  - i.e., it is the absence of catastrophic consequences
- Not always the harm is direct and immediate (e.g. explosion, flood etc.). In the vaccine storage case, violation of temperature boundary would result:
- If detected, in waste of the vaccine
- If not detected, in administering perished vaccine

- The variable *temp* denotes temperature in the cold chamber. How do you formulate safety property?

Safety

- General definition of safety:
- *Safety is a property of the system to not cause harm to its users and environment,*
  - *i.e., it is the absence of catastrophic consequences*

- Not always the harm is direct and immediate. In the vaccine storage case, a violation of temperature boundary would result:
  - If detected, in waste of the vaccine
  - If not detected, in administering perished vaccine

- The variable *temp* denotes temperature in the cold chamber. How do you formulate safety property?

$$temp \leq -70$$

On defining safety property

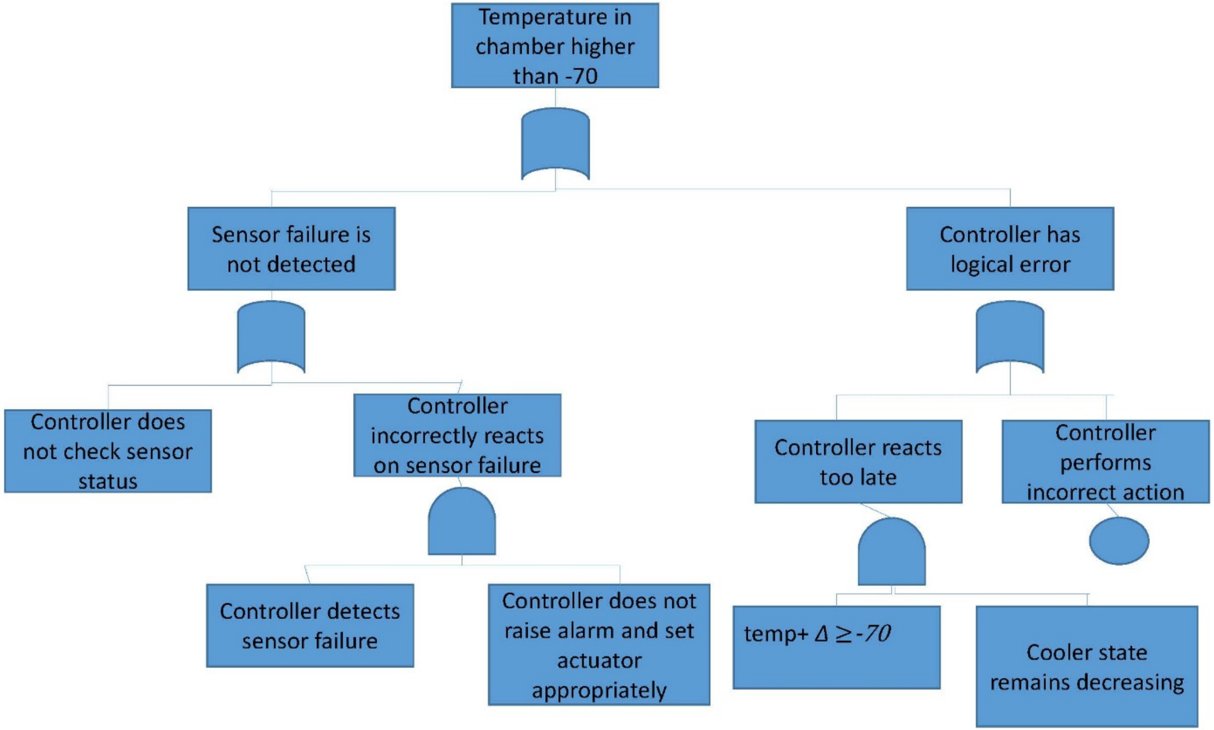
- Our definition of safety property is in terms of real physical temperature
- However, temperature is measured by a sensor.
- Healthy, i.e., correctly working sensor has a certain impresicion Δ
- Reformulating safety property $temp + \Delta \leq -70$

- Can we assume that the sensor is always healthy? Typically no.
- Can we assume that the controlling software always functions correctly, i.e., preserves safety?
- How to deal with various aspects systematically?

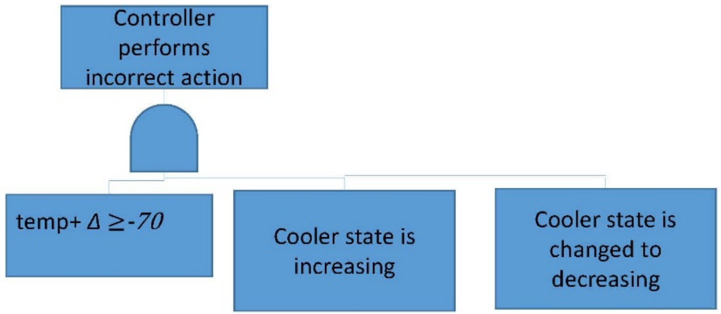
A brief overview of fault trees

- Fault tree is a deductive safety analysis technique
- Fault tree consists of events and logical gates (in the simplest case OR and AND gates)
- It defines the combination of the events that lead to a hazard – undesirable event violating safety requirement
- Fault trees are constructed top-down: we start from the event that we want to avoid and analyse the factors that can contribute to its occurrence

Fault tree for our example



Fault tree for our example cnt.



On defining safety property

- Our definition of safety property is in terms of real physical temperature
- However, temperature is measured by a sensor.
- Healthy, i.e., correctly working sensor has a certain impresicion Δ
- Reformulating safety property $temp + \Delta \leq -70$

- We need to define how the health of the sensor is checked and what system should do to react on failure.

- In a simple case, the sensor produces its health status together with the measurement.

- According to our fault tree, if sensor health is OK then the controller relies on the measurement. If not then raises alarm (failsafe system)

Defining safety property in presence of failures

- We want to express the following:
- If sensor is OK then set the actuator according to the measurement
- If sensor is not OK then set the actuator to safe state and raise alarm
- We need to define the additional variables to represent the sensor status and alarm
- Additional variables:
- sensor: OK, NOT
- alarm: ON, OFF

Specifying system behaviour with sensor failure (informally)

INIT: phase:= plant; cooler := decr; temp :=-70; sensor := OK; alarm := OFF

do infinitely

IF phase= plant AND cooler= incr THEN temp := temp -0.1; phase := cnt

IF phase= plant AND cooler= decr THEN temp := temp +0.1; phase := cnt

IF phase = cnt AND sensor =OK AND $-71 < temp + \Delta \leq -70$ then cooler := incr; phase := plant

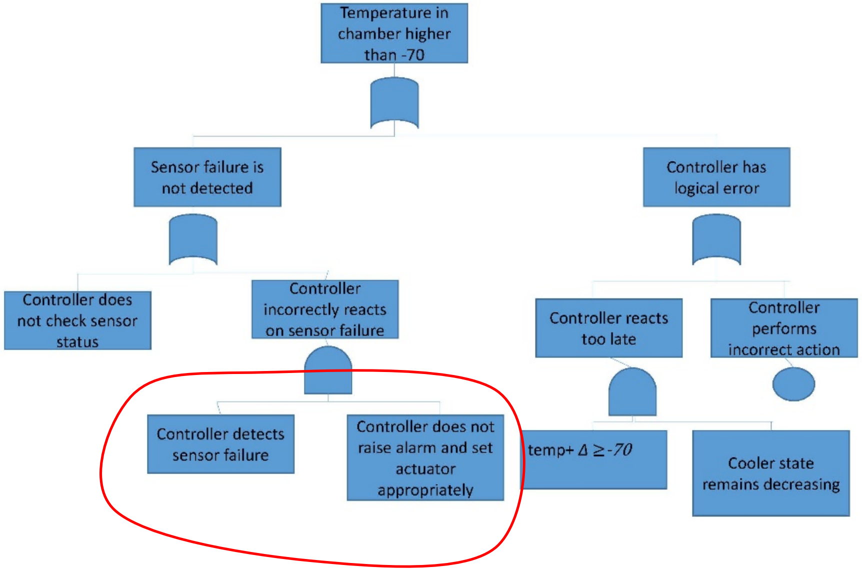
IF phase = cnt AND sensor =OK AND $-71 < temp - \Delta \leq -72$ then cooler := decr; phase := plant

IF phase = cnt AND sensor =NOK then cooler := decr; alarm := ON

enddo

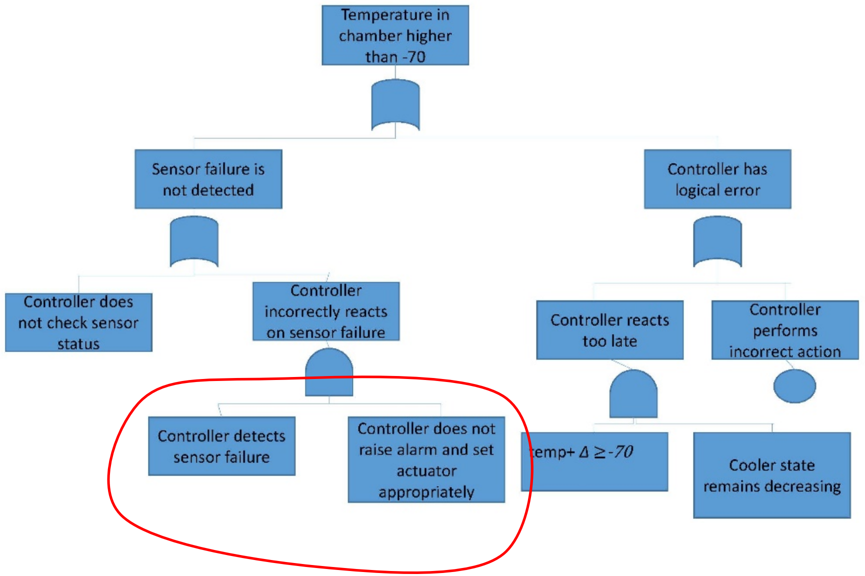
Observe: we made the decision, that predefined safe state of the cooler is decr. After alarm goes ON the system deadlocks, (phase is not changed).

How to verify safety?



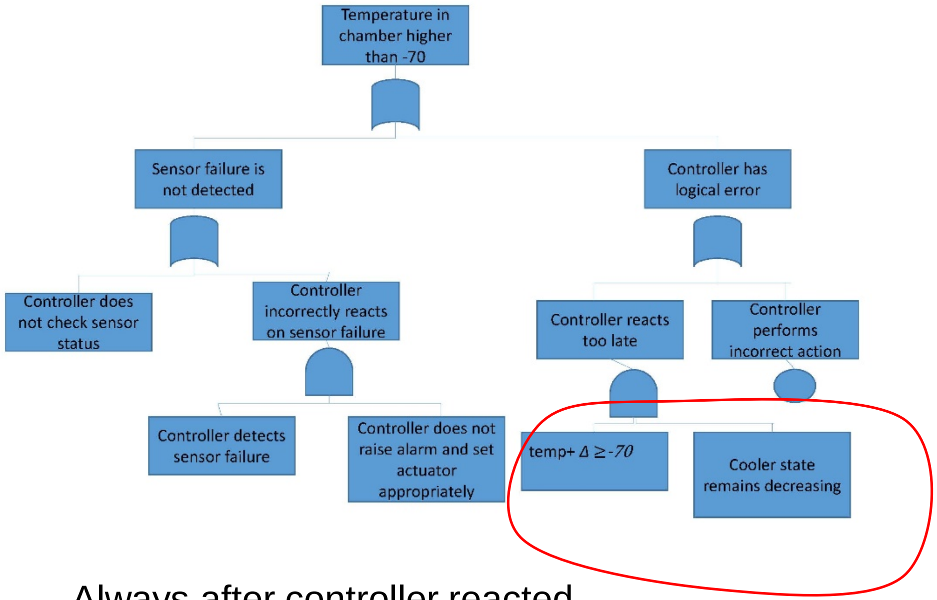
How to express it, so it can be verified?

How to verify safety?



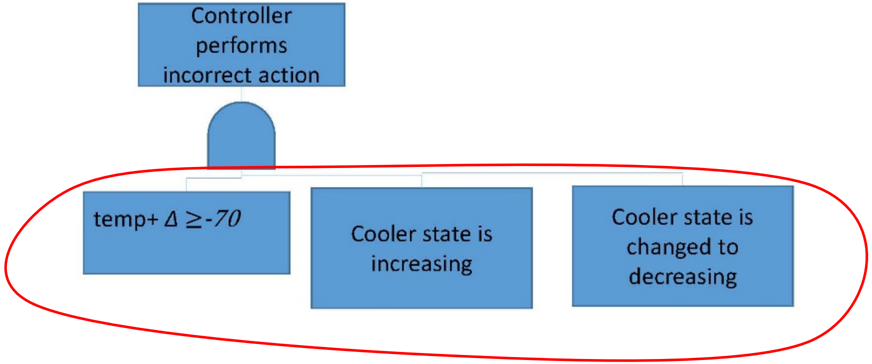
How to express it, so it can be verified?
Always after controller reacted
if sensor is not OK then alarm is raised and actuator is in decr

Fault tree for our example



Always after controller reacted
if sensor is OK and temp + $\Delta \geq$ -70 then cooler is in decr

Fault tree for our example cnt.



Always after controller reacted
if sensor is OK and temp + $\Delta \geq -70$ then cooler is in decr

How to verify safety?

- ”Always” in our expression means that it is an invariant property
- Testing after each statement? For large programms it is unfeasible
- Formal modelling and verification offers a solution: defining an invariant property as a part of the specification of the behaviour of the system.
- Invariant holds means that the predicate defining it evaluates to true after the initialisation and after each possible state transition.

Formal specification of safety-critical systems

- The main idea is to establish a link between safety analysis and verification of system model
- Safety requirements should be reflected in the model: behaviour, invariant
- Formal modelling framework should support verification of the invariant
- For large-scale systems: unfeasible without automatic support for the verification

- Next we will investigate one of the existing specification frameworks – Event-B.

Event-B

- It provides us with a rich modelling language, based on set theory
  - language allows precise descriptions of intended system behaviour (models) to be written in an abstract way
- Event-B uses the abstract machine notation as the basis.
- Event-B is successor of the B Method (also known as classical B).

From the B Method to Event-B

- Inventor: Jean-Raymond Abrial (his previous work is Z framework)
- Both classical B and Event-B are based on set theory
- Analyse models using proofs and additionally -- model checking, animation
- Refinement-based development
  - Verify conformance between higher-level and lower-level models
  - Chain of refinements

o Commercial tools for classical B: Atelier-B (ClearSy, France), B-Toolkit (B-Core, UK)

o Why Event-B: realisation that it is important to reason about system behaviour, not just software

o Event-B is intended for modelling and refining system behaviour

Industrial uses of Event-B

- Event-B in railway interlocking
  - Alstrom, Systerel
- Event-B in smart grids
  - Selex, Critical Software
- Event-B in a cruise control system and a start-stop system
  - Bosch
- Event-B in train control and signaling systems
  - Siemens Transportation

Rodin

- Rodin – the automated tool platform for Event-B.
- www.event-b.org
- Integrated development environment for Event-B
- Models can be created using built-in editor.
- The platform generates proof obligations that can be discharged either automatically or interactively.
- Rodin is a modular software and many extensions are available.
  - These include alternative editors, document generators, team support, and extensions (called plugins) some of which include support decomposition and records.

Wrap-up

- We discussed what is the formal specification and what are the benefits of formal modelling
- We studied a generic architecture of a safety-critical system and performed a high-level safety analysis
- We have outlined (informally) the main principles of modelling a safety-control system and defining safety invariant

- Next lecture is a detailed introduction into Event-B specification language
- First assignment: familarise yourself with Rodin platform by creating and verifying a simple specification
- The rest of the module: more modelling examples, refinement, verification of safety and modelling impact of security on safety

KTH ROYAL INSTITUTE
OF TECHNOLOGY

Questions?