**Industrial Automation**
*Automation Industrielle*
*Industrielle Automation*

# Dependable Software
*Logiciel fiable*
Verlässliche Software

Dr. Jean-Charles Tournier
CERN, Geneva, Switzerland

Material initially prepared by Pr. Dr. H. Kirrmann

## Overview Dependable Software

1. Requirements on Software Dependability
   - –Failure Rates
   - –Physical vs. Design Faults
2. Software Dependability Techniques
   - –Fault Avoidance and Fault Removal
   - –On-line Fault Detection and Tolerance
     - –On-line Fault Detection Techniques
     - –Recovery Blocks
     - –N-version Programming
     - –Redundant Data
3. Examples

## Requirements for Safe Computer Systems

Required failure rates according to the standard IEC 61508:

| safety integrity level | control systems [per hour] | protection systems [per operation] |
|---|---|---|
| 4 | $\geq 10^{-9}$ to $< 10^{-8}$ | $\geq 10^{-5}$ to $< 10^{-4}$ |
| 3 | $\geq 10^{-8}$ to $< 10^{-7}$ | $\geq 10^{-4}$ to $< 10^{-3}$ |
| 2 | $\geq 10^{-7}$ to $< 10^{-6}$ | $\geq 10^{-3}$ to $< 10^{-2}$ |
| 1 | $\geq 10^{-6}$ to $< 10^{-5}$ | $\geq 10^{-2}$ to $< 10^{-1}$ |

most safety-critical systems (e.g. railway signalling)

< 1 failure every 10 000 years

< 1 failure every ~10 years

## Software Problems

Did you ever see software that did not fail once in 10 000 years

(i.e. it never failed during your lifetime)?

- First space shuttle launch delayed due to software synchronisation problem, 1981 (IBM).
- Therac 25 (radiation therapy machine) killed 2 people due to software defect leading to massive overdoses in 1986 (AECL).
- Software defect in 4ESS telephone switching system in USA led to loss of $60 million due to outages in 1990 (AT&T).
- Software error in Patriot equipment: Missed Iraqi Scud missile in Kuwait war killed 28 American soldiers in Dhahran, 1991 (Raytheon).
- ... [add your favourite software bug].

# The Patriot Missile Failure

**The Patriot Missile failure in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors.**
On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

A report of the General Accounting office, GAO/IMTEC-92-26, entitled *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia* analyses the causes (excerpt):

*"The range gate's prediction of where the Scud will next appear is a function of the Scud's known velocity and the time of the last radar detection.*
*Velocity is a real number that can be expressed as a whole number and a decimal (e.g., 3750.2563...miles per hour).*
*Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer or whole number (e.g., 32, 33, 34...).*
*The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. Because of the way the Patriot computer performs its calculations and the fact that its registers are only 24 bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. This conversion results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is directly proportional to the target's velocity and the length of the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the center of the target, making it less likely that the target, in this case a Scud, will be successfully intercepted."*

## Ariane 501 failure

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report.

http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html
(no more available at the original site)

*"The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system.*
*The internal SRI* software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. "*
*SRI stands for Système de Référence Inertielle or Inertial Reference System.

Code was reused from the Ariane 4 guidance system. The Ariane 4 has different flight characteristics in the first 30 s of flight and exception conditions were generated on both inertial guidance system (IGS) channels of the Ariane 5. There are some instances in other domains where what worked for the first implementation did not work for the second.

"Reuse without a contract is folly"
90% of safety-critical failures are requirement errors (a JPL study)

# Malaysia Airline 124: Faulty Unit

**BY** Robert N. Charette // December 2009 (IEEE Spectrum, February 2010)

The passengers and crew of Malaysia Airlines Flight 124 were just settling into their five-hour flight from Perth to Kuala Lumpur that late on the afternoon of 1 August 2005. Approximately 18 minutes into the flight, as the Boeing 777-200 series aircraft was climbing through 36 000 feet altitude on autopilot, the aircraft—suddenly and without warning—pitched to 18 degrees, nose up, and started to climb rapidly. As the plane passed 39 000 feet, the stall and overspeed warning indicators came on simultaneously—something that's supposed to be impossible, and a situation the crew is not trained to handle.

At 41 000 feet, the command pilot disconnected the autopilot and lowered the airplane's nose. The auto throttle then commanded an increase in thrust, and the craft plunged 4000 feet. The pilot countered by manually moving the throttles back to the idle position. The nose pitched up again, and the aircraft climbed 2000 feet before the pilot regained control.

The flight crew notified air-traffic control that they could not maintain altitude and requested to return to Perth. The crew and the 177 shaken but uninjured passengers safely returned to the ground.

The Australian Transport Safety Bureau investigation discovered that the air data inertial reference unit (ADIRU)—which provides air data and inertial reference data to several systems on the Boeing 777, including the primary flight control and autopilot flight director systems—had two faulty accelerometers. One had gone bad in 2001. The other failed as Flight 124 passed 36 571 feet.

The fault-tolerant ADIRU was designed to operate with a failed accelerometer (it has six). The redundant design of the ADIRU also meant that it wasn't mandatory to replace the unit when an accelerometer failed.

**However, when the second accelerometer failed, a latent software anomaly allowed inputs from the first faulty accelerometer to be used, resulting in the erroneous feed of acceleration information into the flight control systems. The anomaly, which lay hidden for a decade, wasn't found in testing because the ADIRU's designers had never considered that such an event might occur.**

The Flight 124 crew had fallen prey to what psychologist Lisanne Bainbridge in the early 1980s identified as the ironies and paradoxes of automation. The irony, she said, is that the more advanced the automated system, the more crucial the contribution of the human operator becomes to the successful operation of the system. Bainbridge also discusses the paradoxes of automation, the main one being that the more reliable the automation, the less the human operator may be able to contribute to that success. Consequently, operators are increasingly left out of the loop, at least until something unexpected happens. Then the operators need to get involved quickly and flawlessly, says Raja Parasuraman, professor of psychology at George Mason University in Fairfax, Va., who has been studying the issue of increasingly reliable automation and how that affects human performance, and therefore overall system performance.

"There will always be a set of circumstances that was not expected, that the automation either was not designed to handle or other things that just cannot be predicted," explains Parasuraman. So as system reliability approaches—but doesn't quite reach—100 percent, "the more difficult it is to detect the error and recover from it," he says.

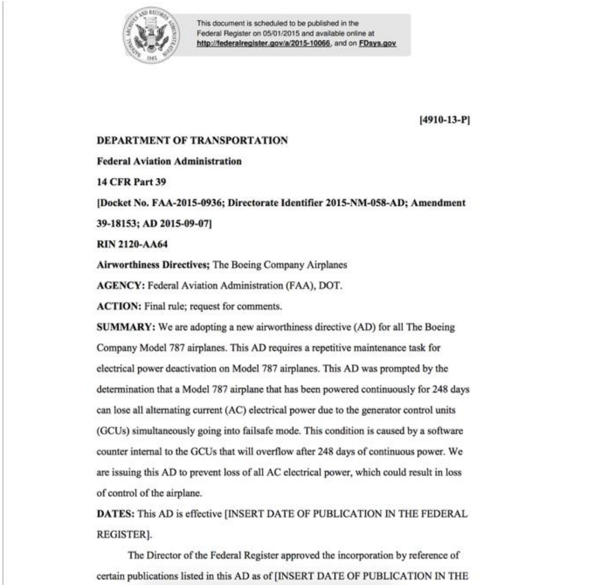
And when the human operator can't detect the system's error, the consequences can be tragic.

## Boeing 787

- Counter Overflow (2015)
  - Department of Transportation – Federal Aviation Administration

“We have been advised by Boeing of an issue identified during laboratory testing. The software counter internal to the generator control units (GCUs) will overflow after 248 days of continuous power, causing that GCU to go into failsafe mode. If the four main GCUs (associated with the engine mounted generators) were powered up at the same time, after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase.”

https://s3.amazonaws.com/public-inspection.federalregister.gov/2015-10066.pdf



This document is scheduled to be published in the Federal Register on 05/01/2015 and available online at http://federalregister.gov/a/2015-10066, and on FDsys.gov

[4910-13-P]

**DEPARTMENT OF TRANSPORTATION**

**Federal Aviation Administration**

**14 CFR Part 39**

**[Docket No. FAA-2015-0936; Directorate Identifier 2015-NM-058-AD; Amendment 39-18153; AD 2015-09-07]**

**RIN 2120-AA64**

**Airworthiness Directives;** The Boeing Company Airplanes

**AGENCY:** Federal Aviation Administration (FAA), DOT.

**ACTION:** Final rule; request for comments.

**SUMMARY:** We are adopting a new airworthiness directive (AD) for all The Boeing Company Model 787 airplanes. This AD requires a repetitive maintenance task for electrical power deactivation on Model 787 airplanes. This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane.

**DATES:** This AD is effective [INSERT DATE OF PUBLICATION IN THE FEDERAL REGISTER].

The Director of the Federal Register approved the incorporation by reference of certain publications listed in this AD as of [INSERT DATE OF PUBLICATION IN THE

## Mars Pathfinder – Priority Inversion Problem - 1997

http://www.cse.chalmers.se/~risat/Report_MarsPathFinder.pdf

In few days after landing, the spacecraft began experiencing total system resets.

No data was lost, but collection was delayed by a day (critical when considering that the mission estimated lifetime was only a couple of weeks)

The source of the problem was due to priority inversion which subsequently **caused a deadline-miss of a critical task, which was identified by a watchdog timer, and finally, the action in such faulty scenario was to reset the spacecraft**

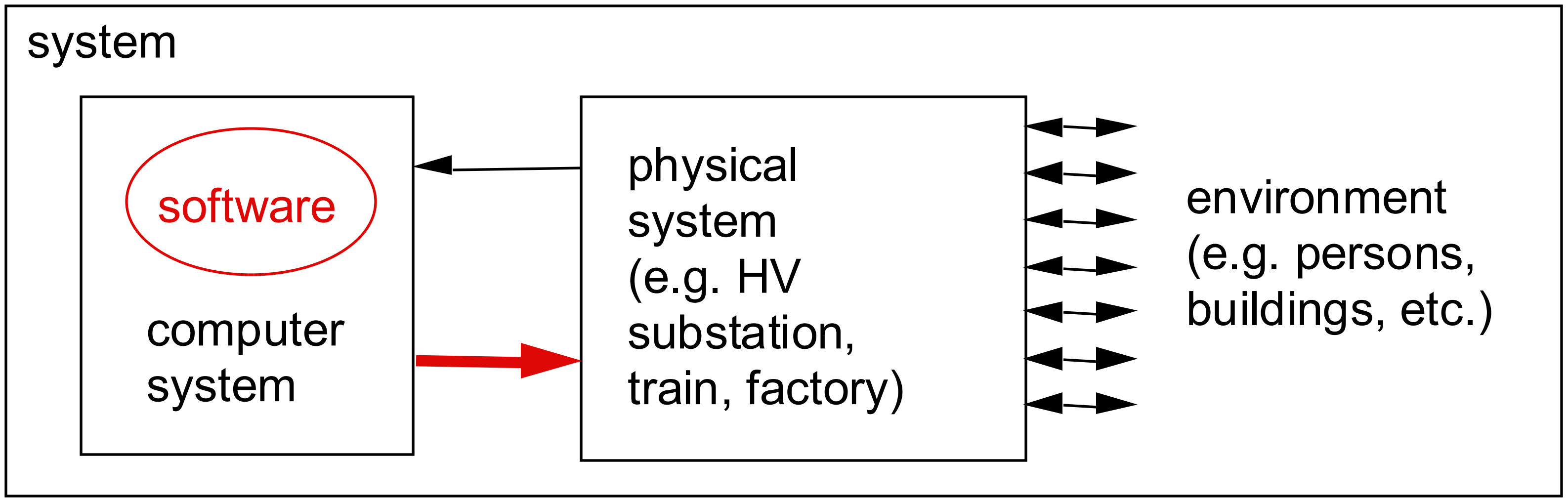
Based on IBM RS6000 processor and WindRiver VxWorks RTOS

## It begins with the specifications ....

A 1988 survey conducted by the United Kingdom's Health & Safety Executive (Bootle, U.K.) of 34 "reportable" accidents in the chemical process industry revealed that inadequate specifications could be linked to 20% (the #1 cause) of these accidents.

## Software and the System

"Software by itself is never dangerous, safety is a system characteristic."


system
software
computer system
physical system (e.g. HV substation, train, factory)
environment (e.g. persons, buildings, etc.)

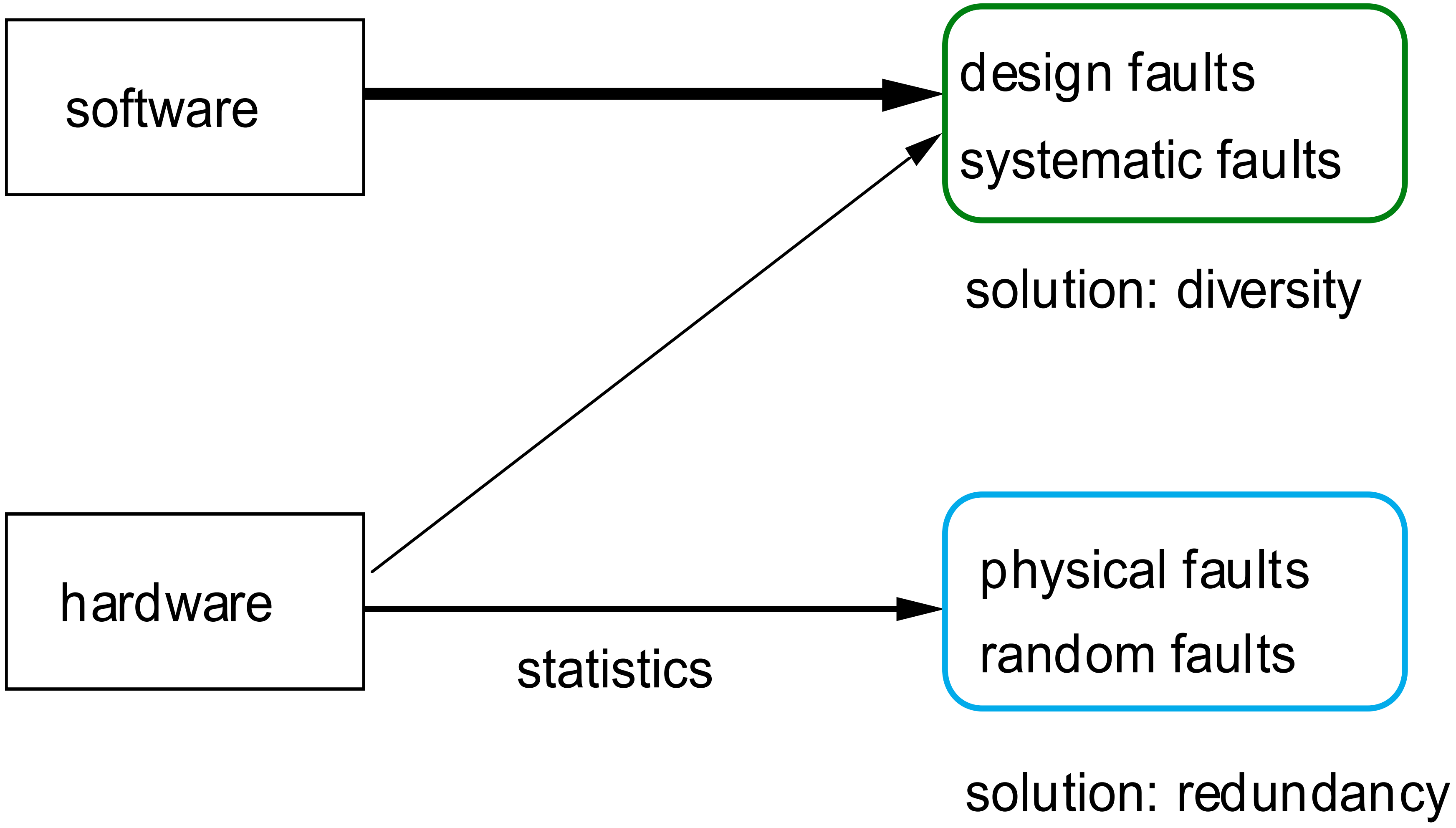
Fault detection helps: if physical system has a safe state (fail-safe system).

Fault tolerance helps: if physical system has no safe state.

Persistency: Computer always produces output (which may be wrong).

Integrity: Computer never produces wrong output (maybe no output at all).

## Which Faults?



## Software Dependability Techniques

1) Against design faults

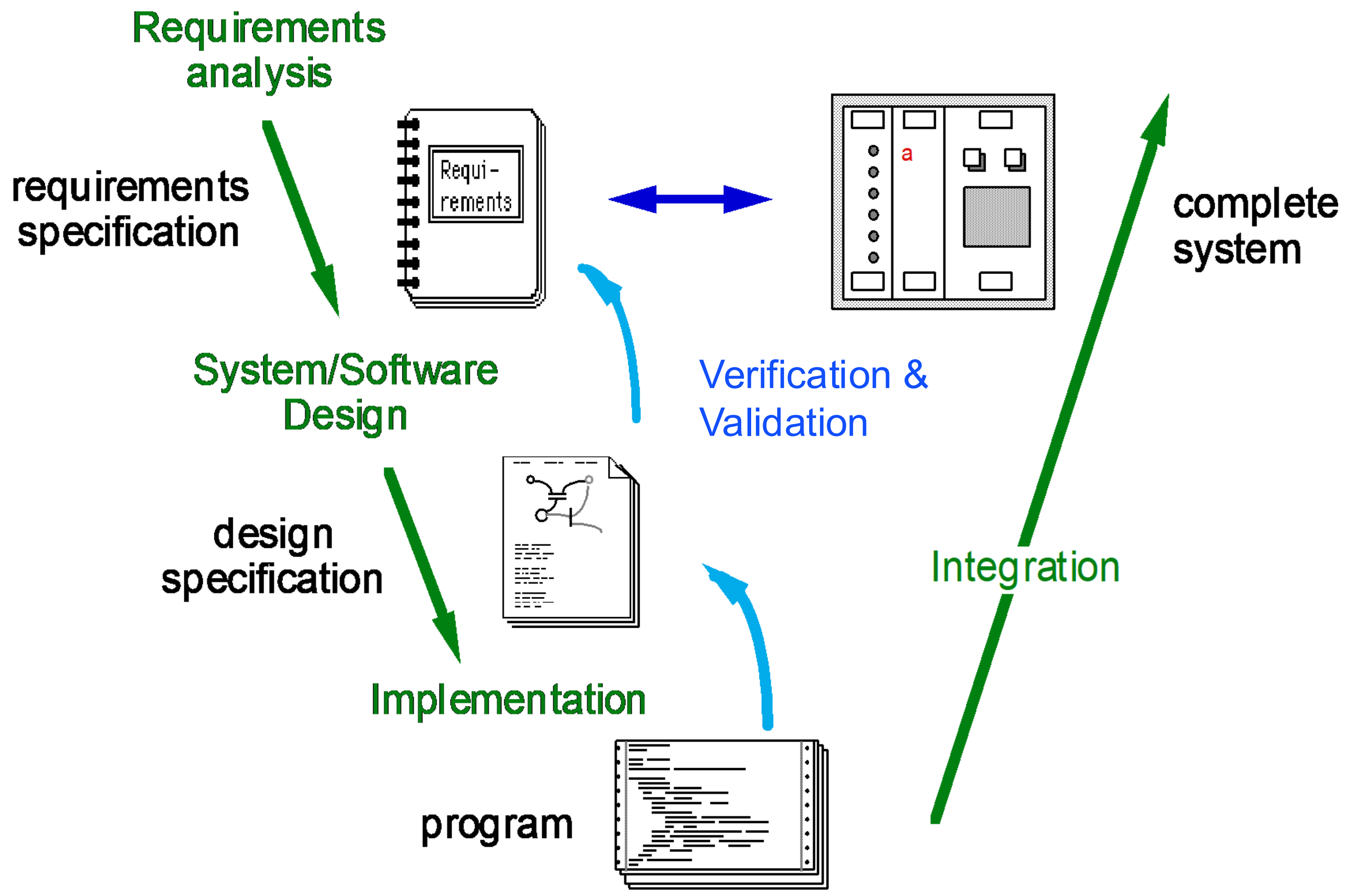
- Fault avoidance → (formal) software development techniques
- Fault removal → verification and validation (e.g. test, formal verification)
- On-line error detection → plausibility checks
- Fault tolerance → design diversity

2) Against physical faults

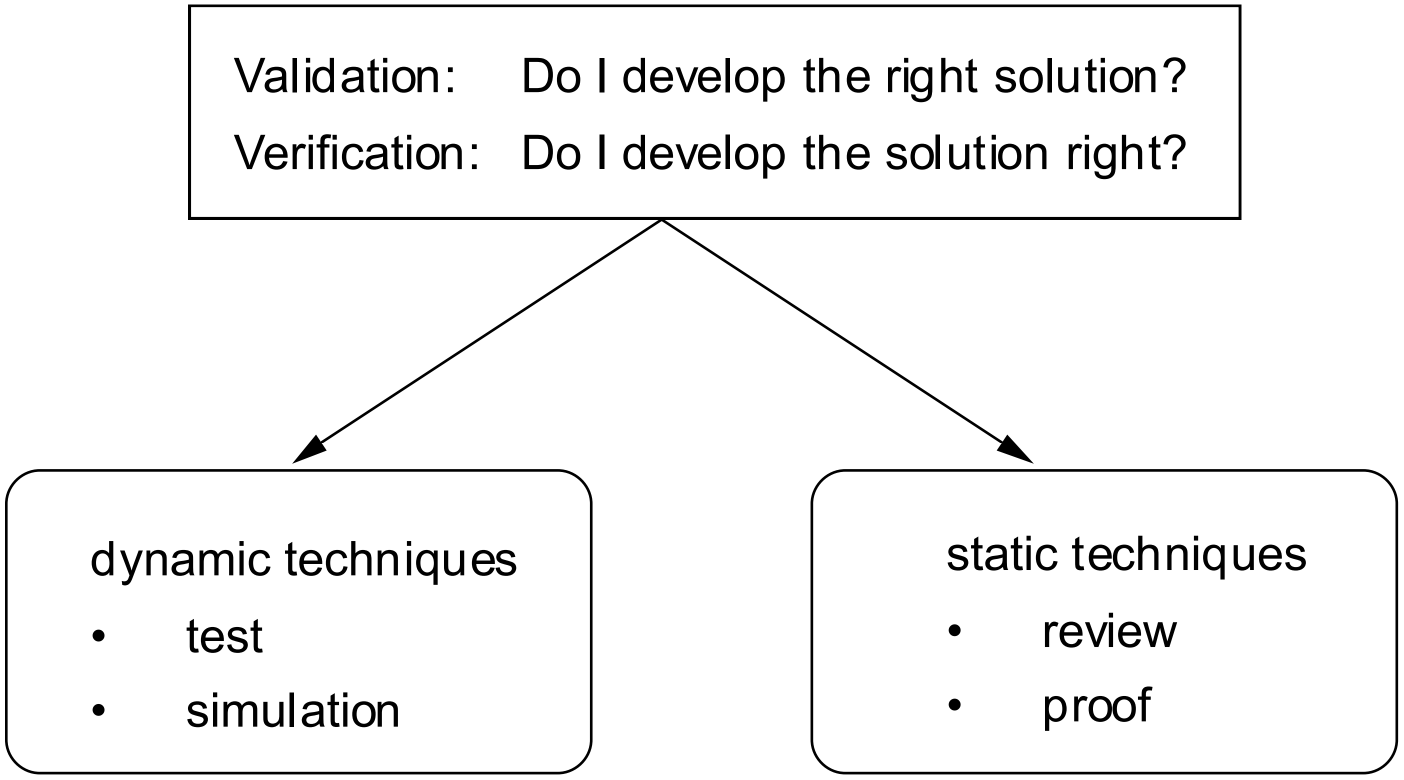
–Fault detection and fault tolerance
(physical faults can not be detected and removed at design time)

- Systematic software diversity (random faults definitely lead to different errors in both software variants)
- Continuous supervision (e.g. coding techniques, control flow checking, etc.)
- Periodic testing

## Fault Avoidance and Fault Removal



## Validation and Verification (V&V)



Validation: Do I develop the right solution?

Verification: Do I develop the solution right?

dynamic techniques
- test
- simulation

static techniques
- review
- proof

## ISO 8402 definitions Validation – Verification

**Validation** := „Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled.“

Validation is the activity of demonstrating that the safety-related system under consideration, before or after installation, meets in all respects the safety requirements specification for that safety-related system. Therefore, for example, software validation means confirming by examination and provision of objective evidence that the software satisfies the software safety requirements specification.

**Verification** := „Confirmation by examination and provision of objective evidence that the specific requirements have been fulfilled.“

Verification activities include:
reviews on outputs (documents from all phases of the safety lifecycle) to ensure compliance with the objectives and requirements of the phase, taking into account the specific inputs to that phase;
design reviews;
tests performed on the designed products to ensure that they perform according to their specification;
integration tests performed where different parts of a system are put together in a step by step manner and by the performance of environmental tests to ensure that all the parts work together in the specified manner.

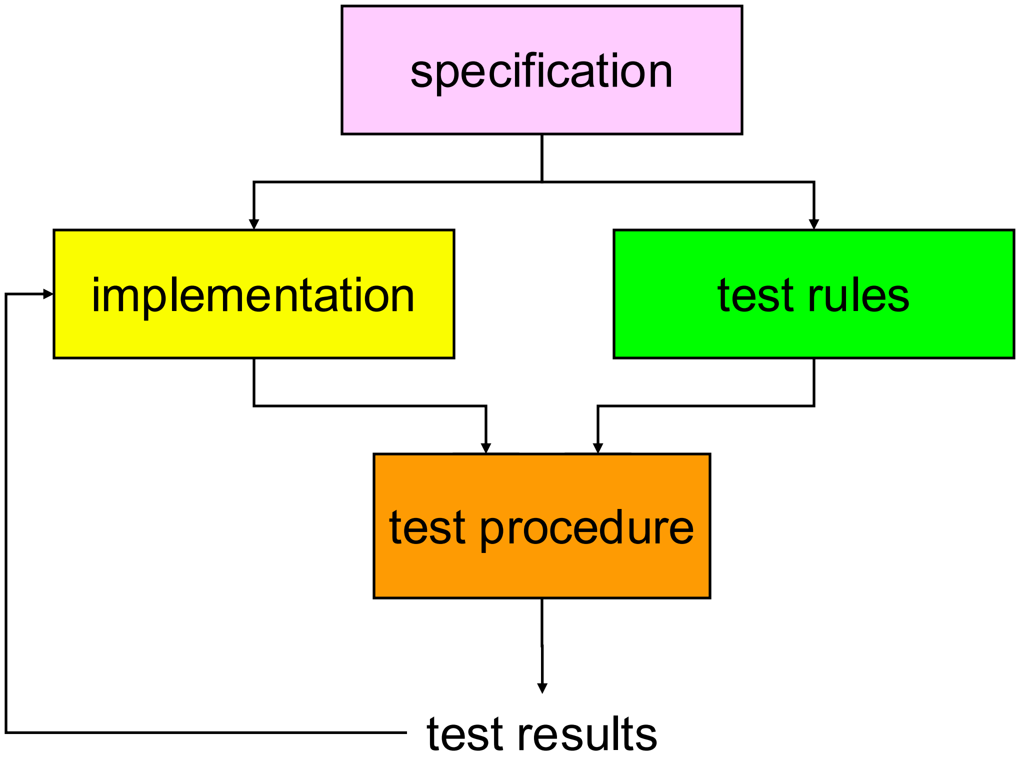
## Verification & Validation

| **Verification** | **Validation** |
|---|---|
| 1. Verification is a static practice of verifying documents, design, code and program. | 1. Validation is a dynamic mechanism of validating and testing the actual product. |
| 2. It does not always involve executing the code. | 2. It always involves executing the code. |
| 3. It is human based checking of documents and files. | 3. It is computer based execution of program. |
| 4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc. | 4. Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc. |
| 5. **Verification** is to check whether the software conforms to specifications. | 5. **Validation** is to check whether software meets the customer expectations and requirements. |
| 6. It can catch errors that validation cannot catch. It is low level exercise. | 6. It can catch errors that verification cannot catch. It is High Level Exercise. |
| 7. Target is requirements specification, application and software architecture, high level,complete design, and database design etc. | 7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product. |
| 8. Verification is done by QA team to ensure that the software is as per the specifications in the SRS document. | 8. Validation is carried out with the involvement of testing team. |
| 9. It generally comes first-done before validation. | 9. It generally follows **verification**. |

http://testingbasicinterviewquestions.blogspot.fr/2012/01/difference-between-verification-and.html

## Testing

Testing requires a test specification, test rules (suite) and test protocol



Testing can only reveal errors, not demonstrate their absence ! (Dijkstra)
Tests can themselves be faulty too…

## Is testing enough?

- System: Elevator controller

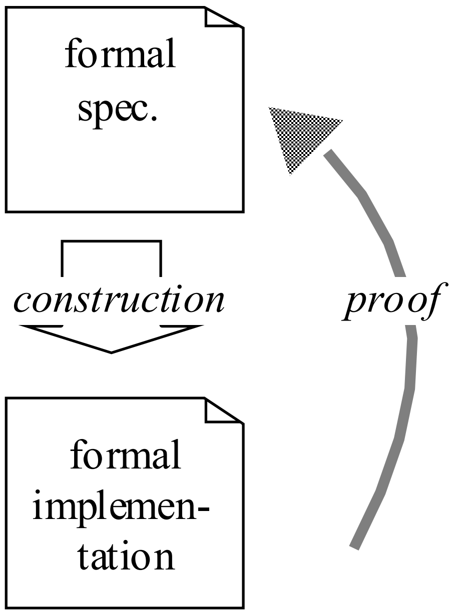
- Testing is useful to validate the system functions
  - If press to 1, goes to floor 1?
  - If press on open door, doors stay open?
  - etc.

- But how to test/express safety properties (invariant)?
  - Can the elevator have the door open while moving => is there a state in the controller for which this condition may be true?

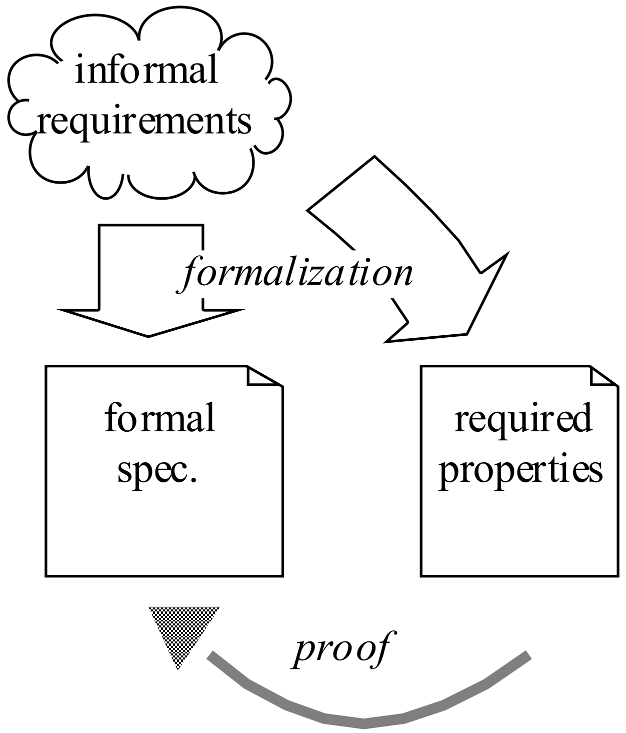
- In addition to testing, formal verification is needed
  - Theorem prover, model checking, etc.

## Formal Proofs

Implementation Proofs



Property Proofs



what is automatically generated need not be tested !
(if you trust the generator | compiler)

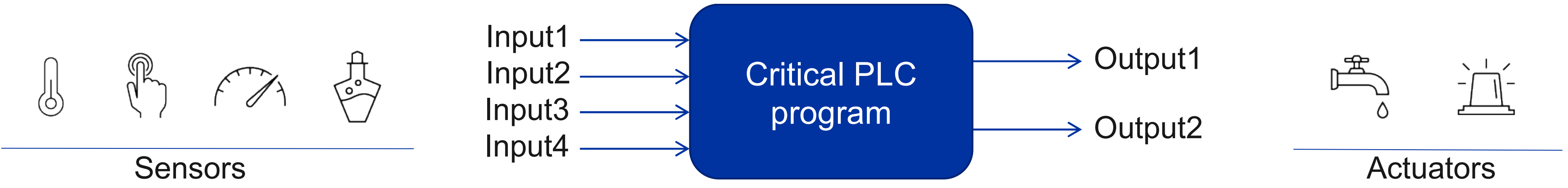
## Formal Languages and Tools

Huge improvement over the last 20 years.
Many tools are now available to non-expert, but formal methods still require a steep learning curve.

| | mathematical foundation | example tools |
|---|---|---|
| **VDM** | dynamic logic (pre- and postconditions) | • Mural from University of Manchester<br>• SpecBox from Adelard |
| **Z** | predicate logic, set theory | • ProofPower from ICL Secure Systems<br>• DST-fuzz from Deutsche System Technik |
| **SDL** | finite-state machines | • SDT from Telelogic<br>• Geode from Verilog |
| **LOTOS** | process algebra | • The LOTOS Toolbox from Information Technology Architecture B.V. |
| **NP** | propositional logic | • NP-Tools from Logikkonsult NP |

c.f. Prof. Kuncak class on Formal Verification
https://edu.epfl.ch/coursebook/en/formal-verification-CS-550
and LARA research group
https://lara.epfl.ch/w/

## Example – CERN PLCverif Model Checking for PLCs



**Functionality requirement**

If **Input1** is False, then **Output2** is False

**Safety requirement**

If **Output1** is True, then **Output2** is False

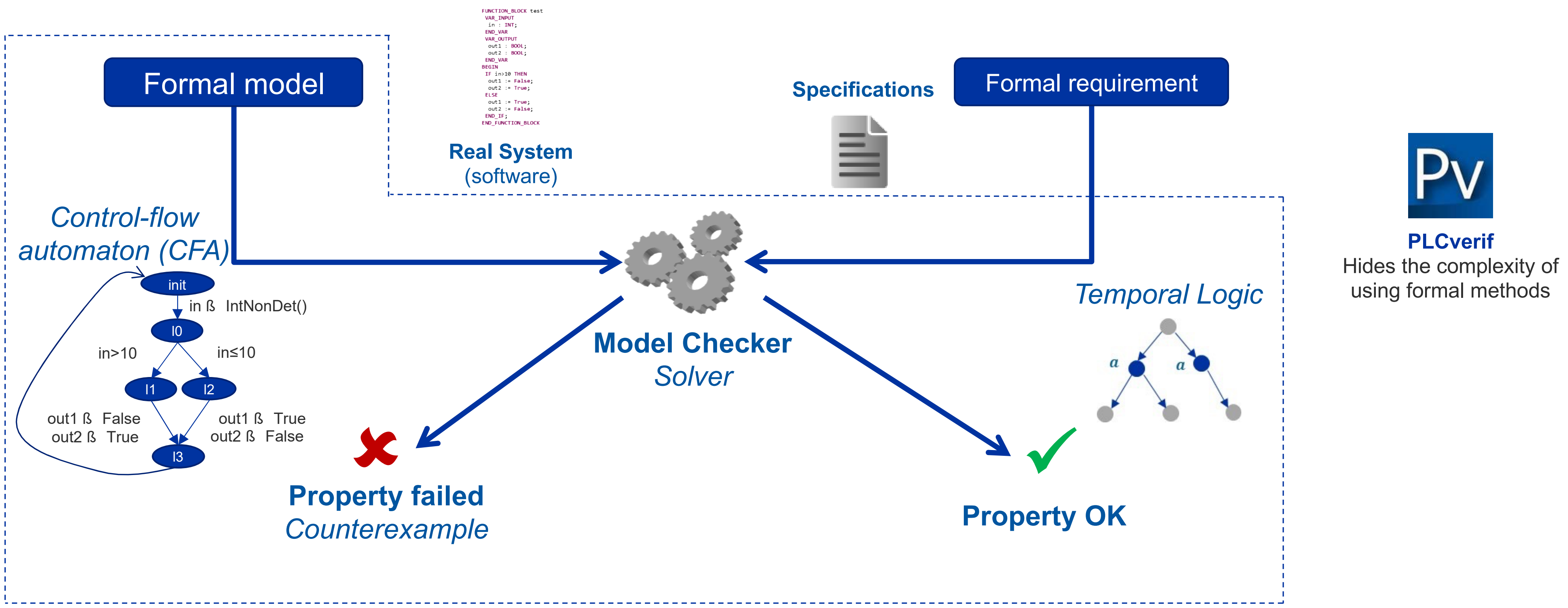
**explore all input combinations**

4 Integer (16-bit) input variables à **$2^{16 \cdot 4} \approx 1.8 \cdot 10^{19}$ combinations** → Solution: **Model checking**

**Problem:** No model checker for PLC programs in industry → Solution: **PLCverif**

https://gitlab.com/plcverif-oss

## PLCverif Worklfow
## Hiding formal method complexity



```
FUNCTION_BLOCK test
 VAR_INPUT
  in : INT;
 END_VAR
 VAR_OUTPUT
  out1 : BOOL;
  out2 : BOOL;
 END_VAR
BEGIN
 IF in>10 THEN
  out1 := False;
  out2 := True;
 ELSE
  out1 := True;
  out2 := False;
 END_IF;
END_FUNCTION_BLOCK
```

## On-line Error Detection by N-Version programming

N-Version programming is the software equivalent of massive redundancy (workby)

"detection of design errors on-line by diversified software, independently programmed in different languages by independent teams, running on different computers, possibly of different type and operating system".

Difficult to ensure that the teams end up with comparable results, as most computations yield similar, but not identical results:

- rounding errors in floating-point arithmetic
  (use of identical algorithms)

- different branches taken at random (synchronize the inputs)
  ```
  if (T > 100.0) {...}
  ```

- equivalent representation (are all versions using the same data formats ?)
  ```
  if (success == 0) {….}
  IF success = TRUE THEN
  int flow = success ?  12: 4;
  ```

Difficult to ensure that the teams do not make the same errors
(common school, and interpret the specifications in the same wrong way)

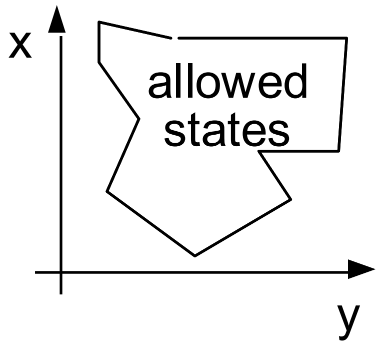
# On-line error detection by Acceptance Tests

Acceptance Test are invariants calculated at run-time

- definition of invariants in the behaviour of the software

- set-up of a "don't do" specification

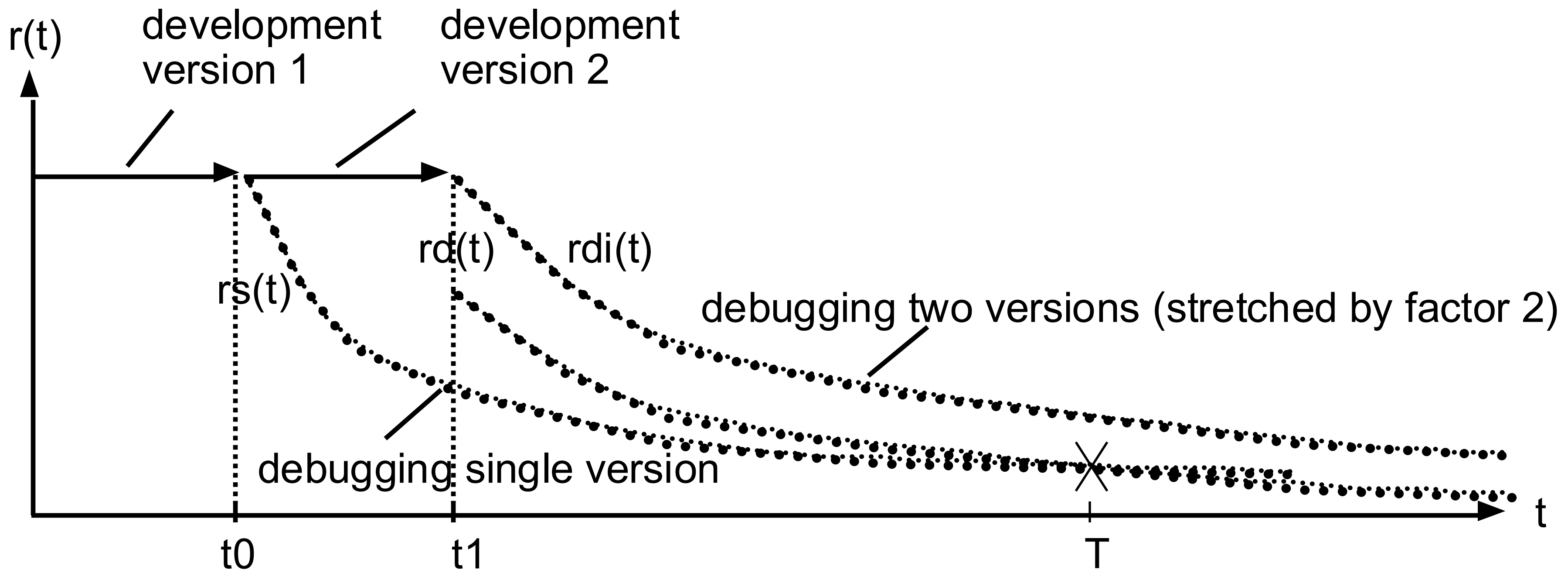
- plausibility checks included by the programmer of the task (efficient but cannot cope with surprise errors).



## Cost Efficiency of Fault Removal vs. On-line Error Detection

Design errors are difficult to detect and even more difficult to correct on-line.
The cost of diverse software can often be invested more efficiently in
off-line testing and validation instead.

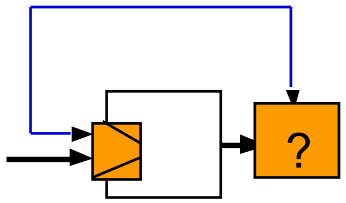
Rate of safety-critical failures (assuming independence between versions):



## On-line Error Detection

- periodical tests

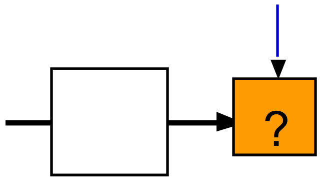
example test



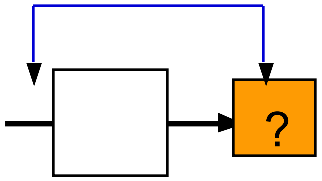
overhead

- continuous supervision

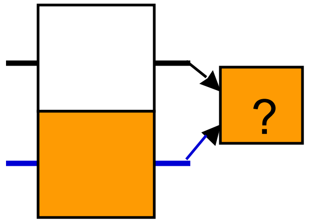
plausibility check



acceptance test



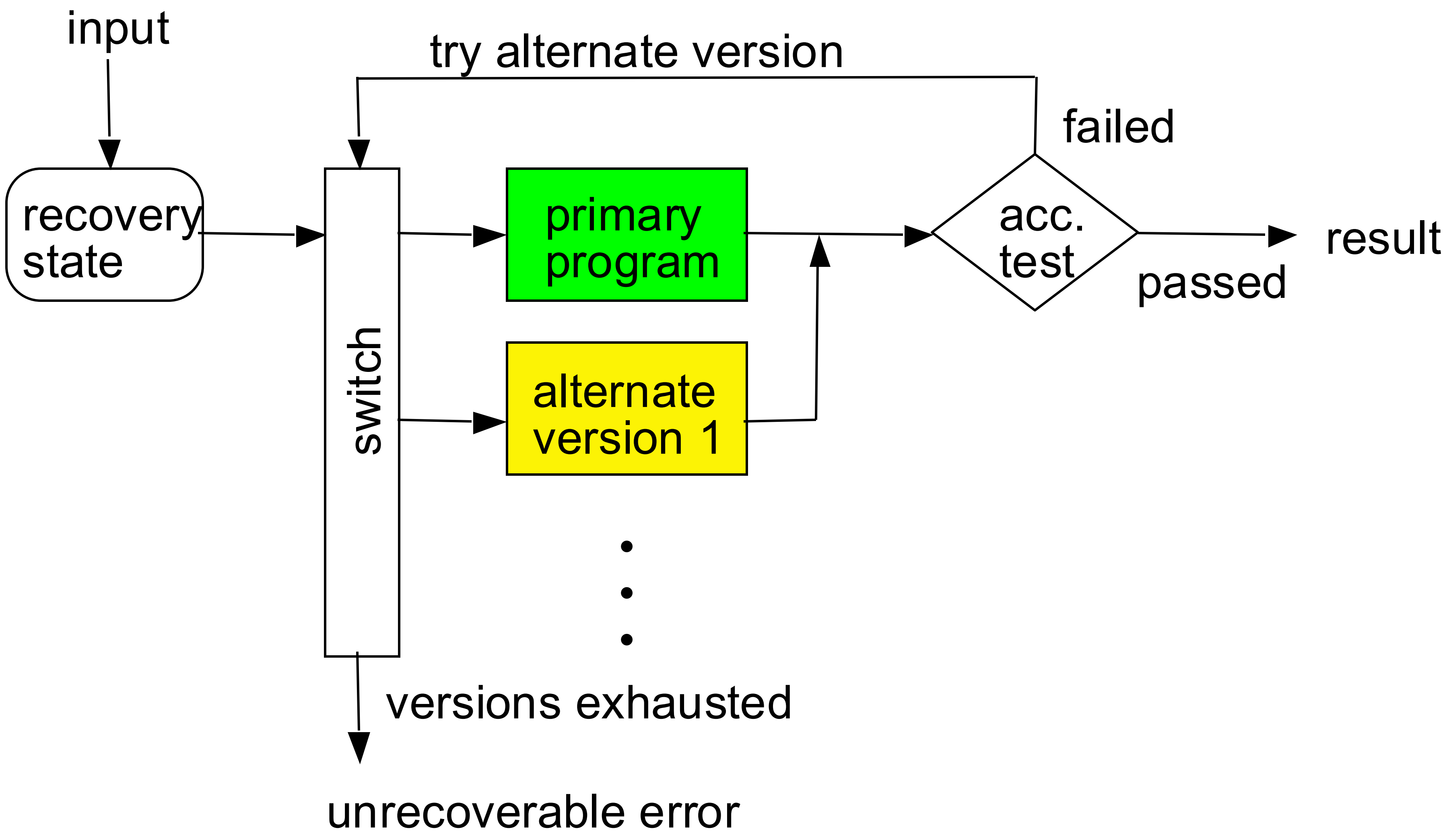
redundancy/diversity
hardware/software/time



## Plausibility Checks / Acceptance Tests

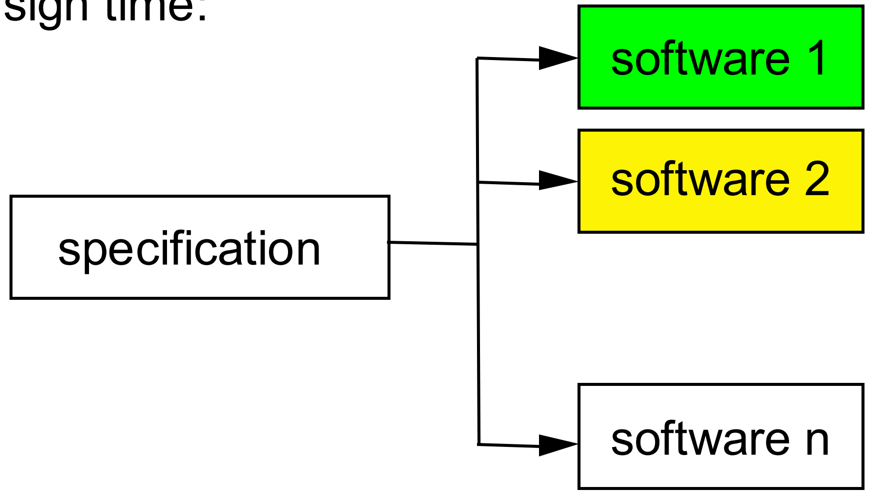
| | |
|---|---|
| • range checks | $0 \le$ train speed $\le 500$ |
| | safety assertions |
| • structural checks | given list length / last pointer NIL |
| • control flow checks | set flag; go to procedure; check flag |
| | hardware signature monitors |
| • timing checks | checking of time-stamps/toggle bits |
| | hardware watchdogs |
| • coding checks | parity bit, CRC |
| • reversal checks | compute $y = \sqrt{x}$; check $x = y^2$ |

## Recovery Blocks



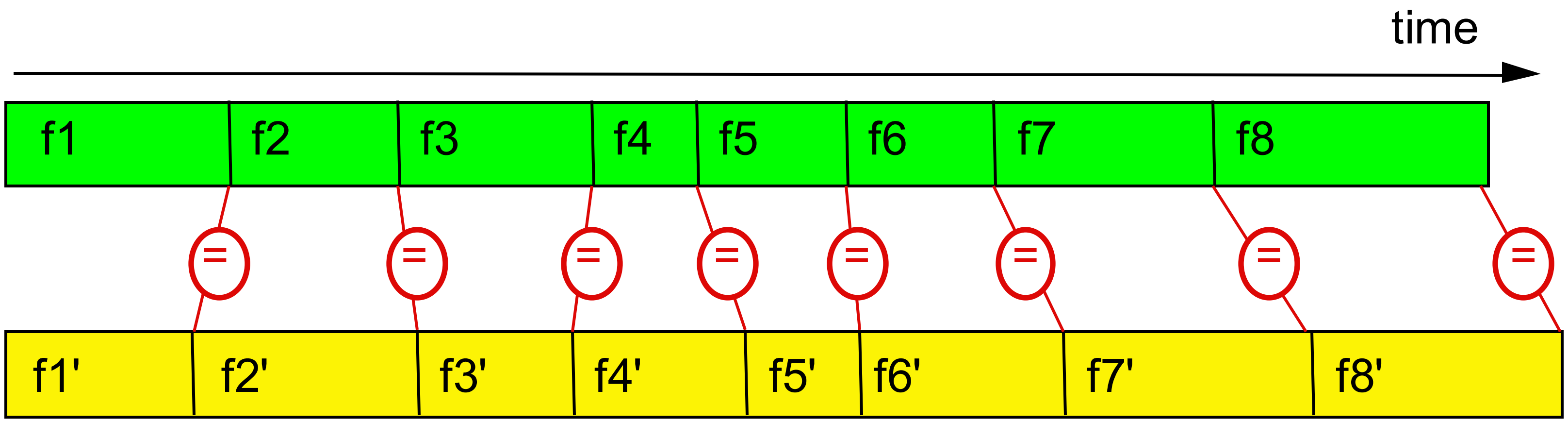
# N-Version Programming (Design Diversity)

design time:



different teams
different languages
different data structures
different operating system
different tools (e.g. compilers)
different sites (countries)
different specification languages
• • •

run time:



## Issues in N-Version Programming

- number of software versions (fault detection ↔ fault tolerance)
- hardware redundancy ↔ time redundancy (real-time !)
- random diversity ↔ systematic diversity
- determination of cross-check (voting) points
- format of cross-check values
- cross-check decision algorithm (consistent comparison problem !)
- recovery/rollback procedure (domino effect !)
- common specification errors (and support environment !)
- cost for software development
- diverse maintenance of diverse software ?

## Consistent Comparison Problem

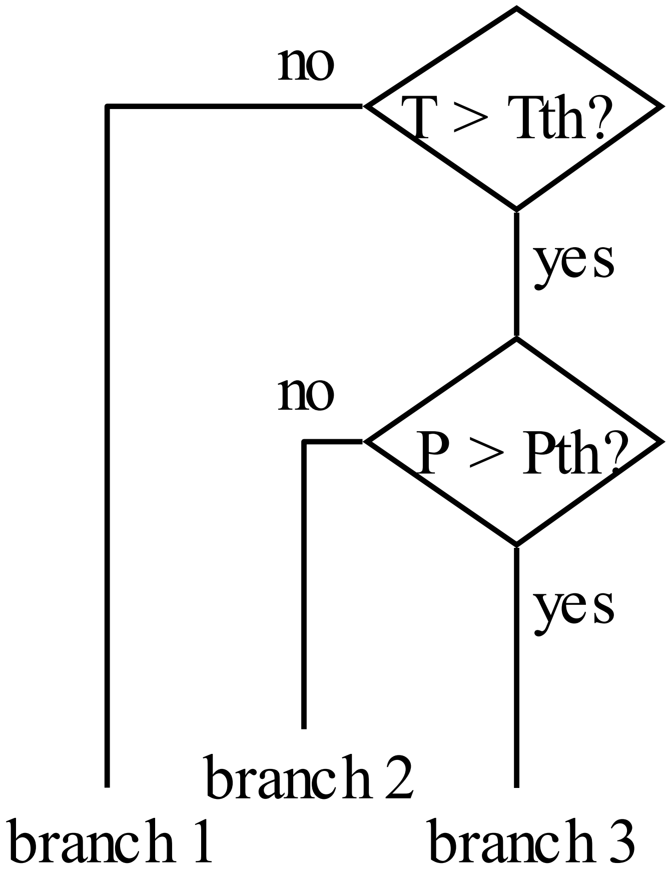
Problem occurs if floating point numbers are used.

Finite precision of hardware arithmetic
→ result depends on sequence of computation steps.

Thus: Different versions may result in slightly different results
→ result comparator needs to do "inexact comparisons"

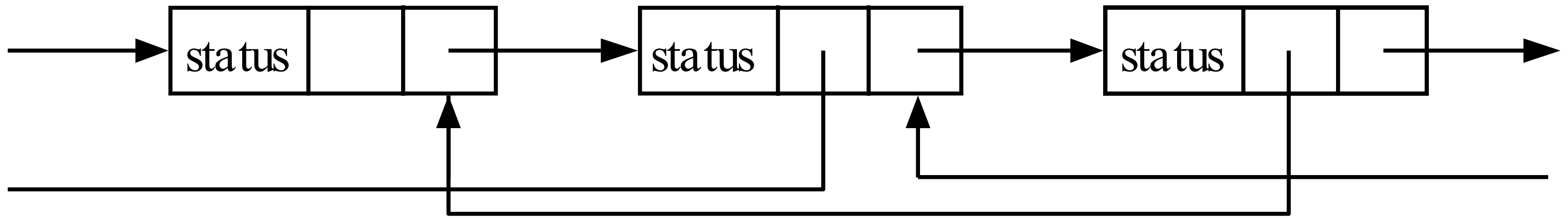
Even worse: Results used internally in subsequent computations with comparisons.

Example: Computation of pressure value P and temperature value T with floating point arithmetic and usage as in program shown:

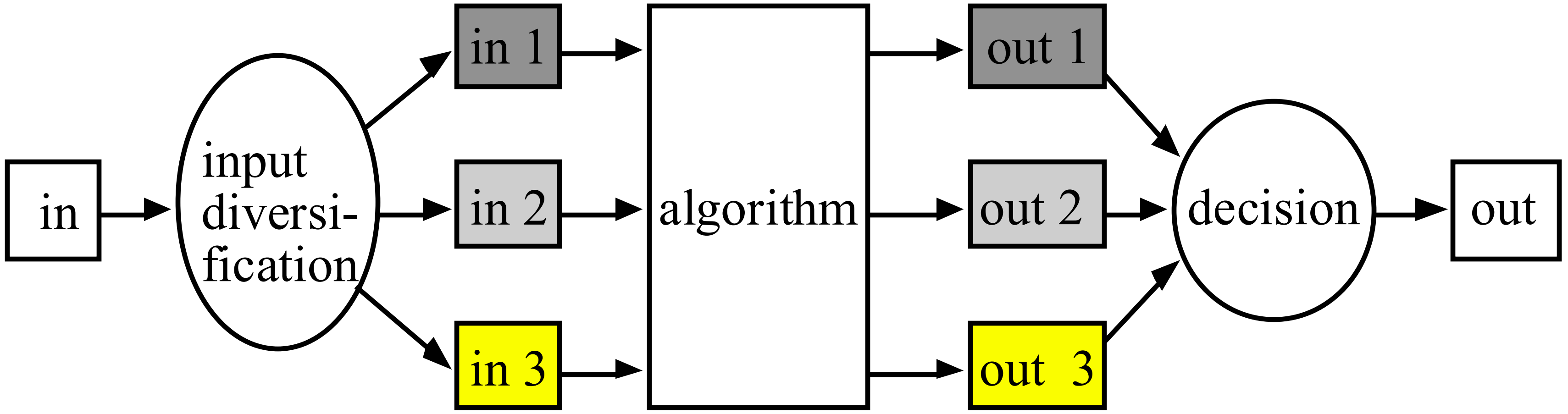


## Redundant Data

Redundantly linked list



Data diversity



## Examples

Use of formal methods

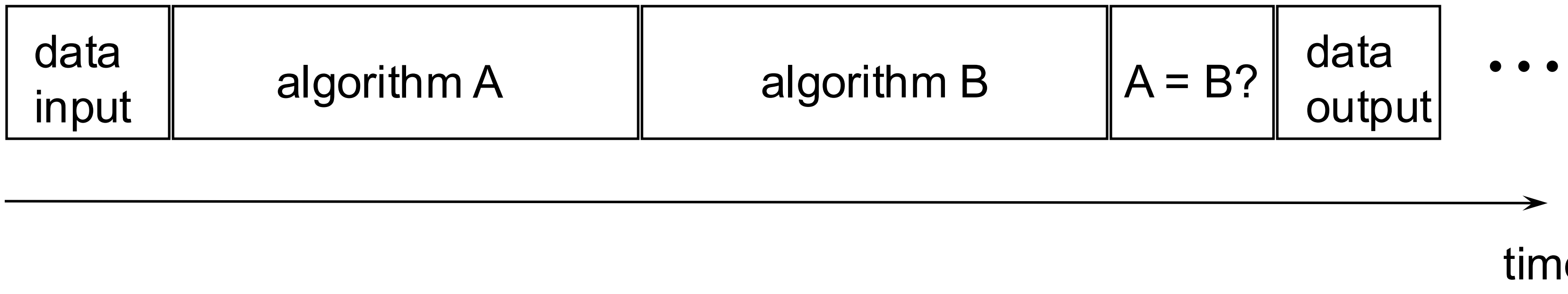
- Formal specification with Z
  Tektronix: Specification of reusable oscilloscope architecture
- Formal specification with SDL
  ABB Signal: Specification of automatic train protection systems
- Formal software verification with Statecharts
  GEC Alsthom: SACEM - speed control of RER line A trains in Paris
- CERN PLCverif tool to formally verify safety PLC programs

Use of design diversity

- 2x2-version programming
  Aerospatiale: Fly-by wire system of Airbus A310
- 2-version programming
  US Space Shuttle: PASS (IBM) and BFS (Rockwell)
- 2-version programming
  ABB Signal: Error detection in automatic train protection system EBICAB 900

## Example: 2-Version Programming (EBICAB 900)

Both for physical faults and design faults (single processor $\rightarrow$ time redundancy).



- 2 separate teams for algorithms A and B
  3rd team for A and B specs and synchronisation
- B data is inverted, single bytes mirrored compared with A data
- A data stored in increasing order, B data in decreasing order
- Comparison between A and B data at checkpoints
- Single points of failure (e.g. data input) with special protection (e.g. serial input with CRC)