

1. Introduction

Jean-Raymond Abrial

2009

- To give you some insights about modelling and formal reasoning
- To show that programs can be correct by construction
- To show that modelling can be made practical
- To illustrate this approach with many examples

- By the end of the course you should be comfortable with:
 - Modelling (versus programming)
 - Abstraction and Refinement
 - Some mathematical techniques used for reasoning
 - The practice of proving as a means to construct programs
 - The usage of some proving tools



- August 10, 1628: The Swedish warship Vasa sank.
- This was her maiden voyage.
- She sailed about 1,300 meters only in Stockholm harbor.
- 53 lives were lost in the disaster.

1. Changing **requirements** (by **King Gustav II Adolf**).
2. Lack of **specifications** (by **Ship Builder Henrik Hybertsson**).
3. Lack of **explicit design** (by **Subcontractor Johan Isbrandsson**)
(No **scientific calculation** of the ship stability)
4. **Test outcome** was not followed (by **Admiral Fleming**)

- Enter keywords "**Vasa disaster**" in Google
- **The Vasa: A Disaster Story with Software Analogies.**
By **Linda Rising**.
The Software Practitioner, January-February 2001.
<http://members.cox.net/risingl1/articles/Vasa.pdf>
- **Why the Vasa Sank: 10 Problems and Some Antidotes for Software Projects.**
By **Richard E. Fairley** and **Mary Jane Willshire**.
IEEE Software, March-April 2003.
http://www.cse.ogi.edu/~dfairley/The_vasa.pdf

- June 4, 1996: The launch vehicle Ariane 5 exploded.
- This was its maiden voyage.
- It flew for about 37 Sec only in Kourou's sky.
- No injury in the disaster.

-
- Normal behavior of the launcher for 36 Sec after lift-off
 - Failure of both Inertial Reference Systems almost simultaneously
 - Strong pivoting of the nozzles of the boosters and Vulcain engine
 - Self-destruction at an altitude of 4000 m (1000 m from the pad)

- Both inertial computers failed because of **overflow on one variable**
- This caused a **software exception** and stops these computers
- These computers sent **post-mortem info** through the bus
- **Normally** the main computer receives **velocity info** through the bus
- The main computer was **confused** and **pivoted the nozzles**

- The faulty program was **working correctly on Ariane 4**
- The faulty program was **not tested for A5** (since it worked for A4)
- But the velocity of Ariane 5 is **far greater than that of Ariane 4**
- The faulty program happened to be **useless for Ariane 5**
- It was kept for **commonality reasons**

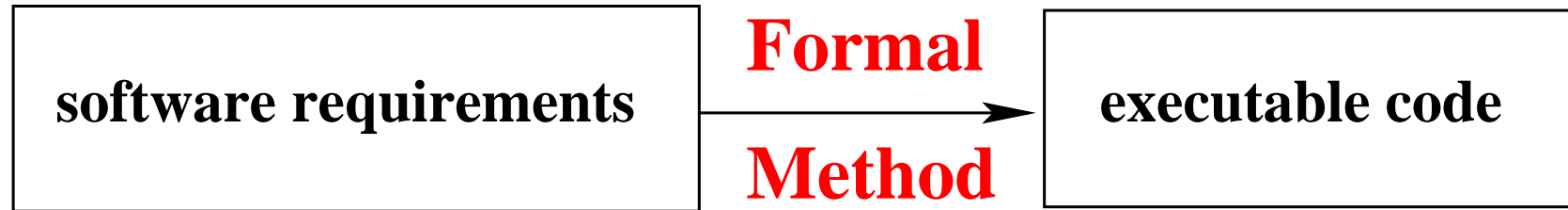
- Enter keywords "flight 501" in Google
- Ariane 5 flight 501 Inquiry Board Report:
<http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>
- INRIA report challenging the Inquiry Board Report:
<ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-3079.pdf>

1. About **formal methods** in general
2. About the **requirement analysis** (example)
3. About **modelling**

1. About **formal methods** in general

- What are they used for?
- When are they to be used?
- Is UML a formal method?
- Are they needed when doing OO programming?
- What is their definition?

- Helping **engineers** in doing the following **transformation**:



- It does not seem to be different from **ordinary programming**

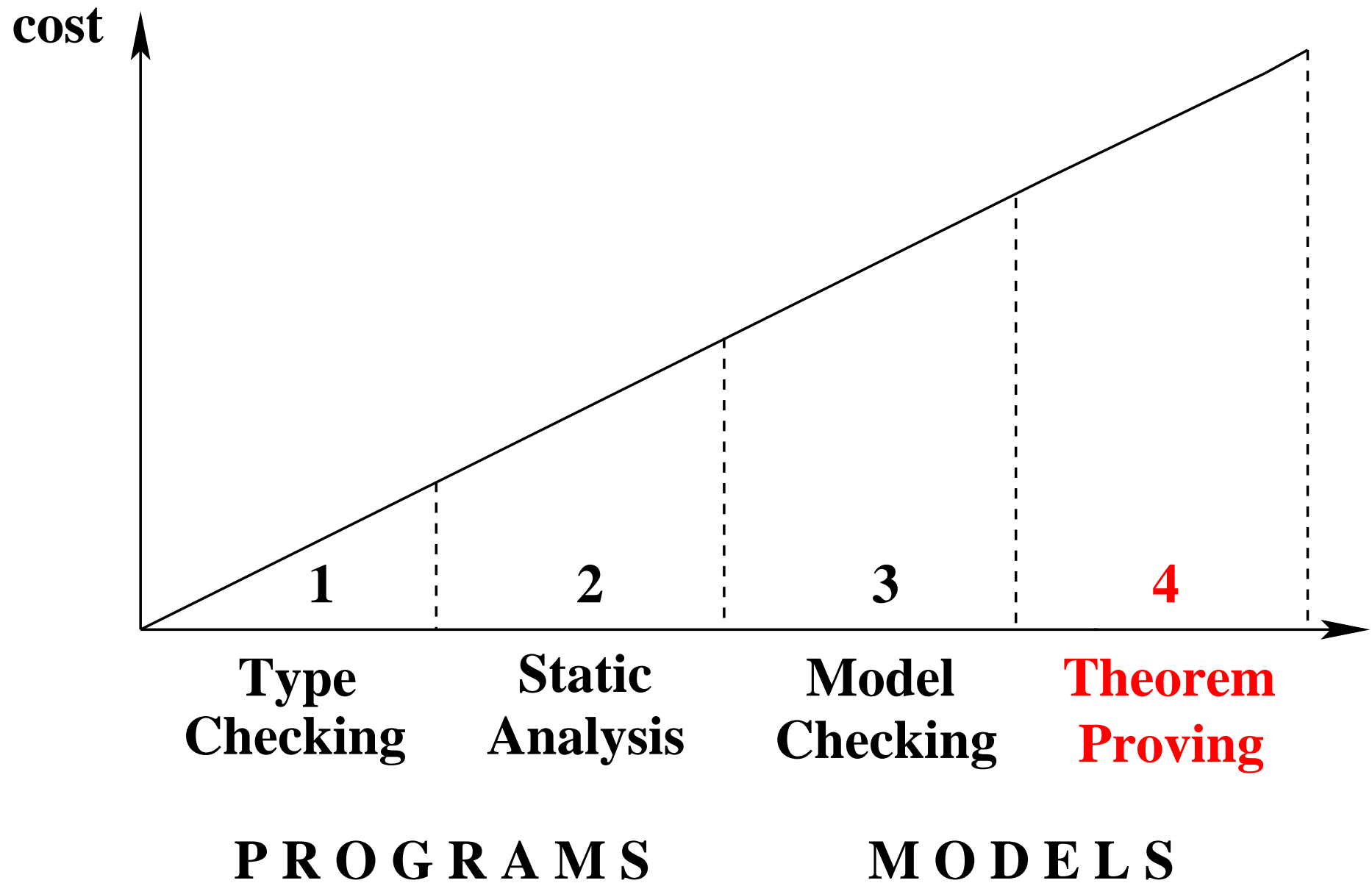
- A formal method is a **systematic approach** used to determine whether a **program has certain wishful properties**
- Different **kinds of formal methods** (according to this definition)
 - Type checking
 - Static analysis
 - Model checking
 - Theorem proving

- Controlling low level **properties of variables** in a program
- A **type** defines:
 - a **set of values** to be assigned to a **variable**
 - the **operations** that can be performed on a variable
 - the way a variable will be **stored in the memory**
- **Type checking** controls that:
 - value **assignments** to a variable is **correct**
 - the variable is used in **authorized operations only**
- It is done **automatically** within the compiler

- It is an **automatic technique** used for checking that a program will not have certain **run-time errors**
- **Typical run-time errors** detected:
 - Division by zero
 - Array bound overflow
 - Arithmetic overflow (floating point)
- The analysis is performed by **abstracting the program variables** and "executing" the **resulting abstraction** rather than the program itself
- The abstract interpretation may lead to **false alarm**

- The properties to be checked are **not properties of programs**
- They are properties of **models of program**
- Usually, these models denote **finite state machines**
(state and transitions)
- The properties to be checked are often **temporal properties**
(reachability)
- Model checkers work **automatically**

- This is the **approach I am going to develop** in this course
- It concentrates on the construction of models by **successive refinements**
- The properties to be proved are parts of the models: **invariants** and **refinement**
- At the end of the process, the most refined model is **automatically translated** into a program



- In 1 and 2, one works on **programs**
- In 3 and 4, one works on **models**
- In 1 and 4, you prove a property that is **part of the object to analyze**
- In 2 and 3, you prove a property that is **proposed externally**

- When there is **nothing better available**.
- When the **risk** is too high (e.g. in **embedded systems**).
- When people have already **suffered enough**.
- When people question their **development process**.
- Decision of using formal methods is **always strategic**.

- It is to be **opposed to a general purpose computer** system like a PC Operating System
- The computer is **encapsulated** within the device it controls
- It is doing **for ever** a number of **specific tasks**
- Examples: Systems controlling
 - a portable telephone
 - an aircraft or a space ship
 - a driverless train
 - a nuclear reactor
 - ...

- Such systems are working in **close connection** with an external often **unpredictable environment** (physical and human)
- **Reliability** is usually very important
- **Error detection and recovery** must be performed (degraded mode)
- **Real-time constraints** have to be taken into account
- Consequently, the **software** has to be developed **with great care**

- This is a **difficult** question.
- Today many formal methods **vendors**.
- "Formal method" has become a meaningless **buzz word**.
- "Formal" alone **does not mean anything**.

- Is there a **theory** behind your **F**ormal **M**ethod with **P**roofs (**FMP**) ?
- What kind of **language** is your FMP using ?
- Does there exist any **refinement** mechanism in your FMP ?
- Have you got an efficient **automatic** prover ?

- You have to be a **mathematician**.
- **Formalism** is hard to master.
- Not **visual** enough (no boxes, arrows, etc.).
- People will **not** be able to do formal **proofs**.

- You have to **think a lot** before final coding.
- Incorporation in **development process**.
- **Model building** is an elaborate activity.
- **Reasoning** by means of **proof** is necessary.
- Poor quality of **requirement documents**.

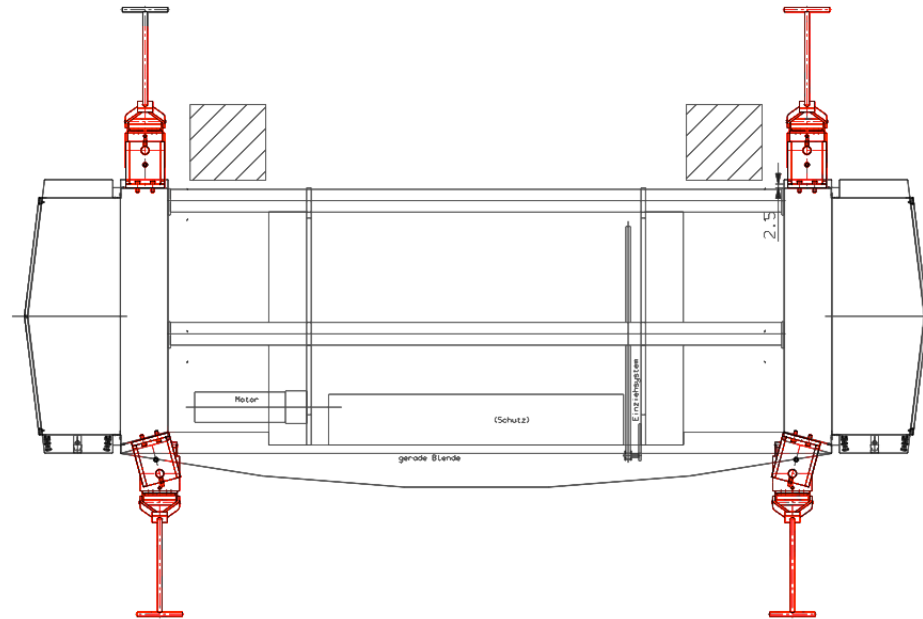
- Some **mature** engineering disciplines:
 - Avionics,
 - Civil engineering,
 - Mechanical engineering,
 - Train systems,
 - Ship building.
- Are there any **equivalent approaches** to Formal Methods with Proofs?
- Yes, **BLUE PRINTS**

- A certain **representation** of the system we want to build
- It is **not a mock-up** (although mock-ups can be **very useful too**)
- The **basis is lacking** (you cannot “drive” the blue print of a car)
- Allows to **reason** about the future system **during its design**
- **Is it important?** (according to professionals) **YES**

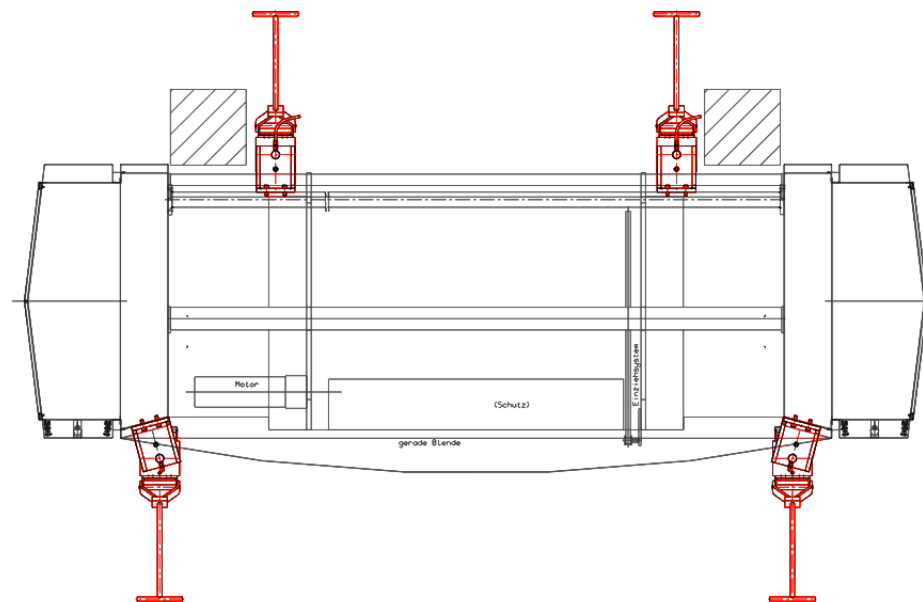
- Defining and calculating its **behavior** (what it does)
- Incorporating **constraints** (what it must not do)
- Defining **architecture**
- Based on some **underlying theories**
 - strength of materials,
 - fluid mechanics,
 - gravitation,
 - etc.

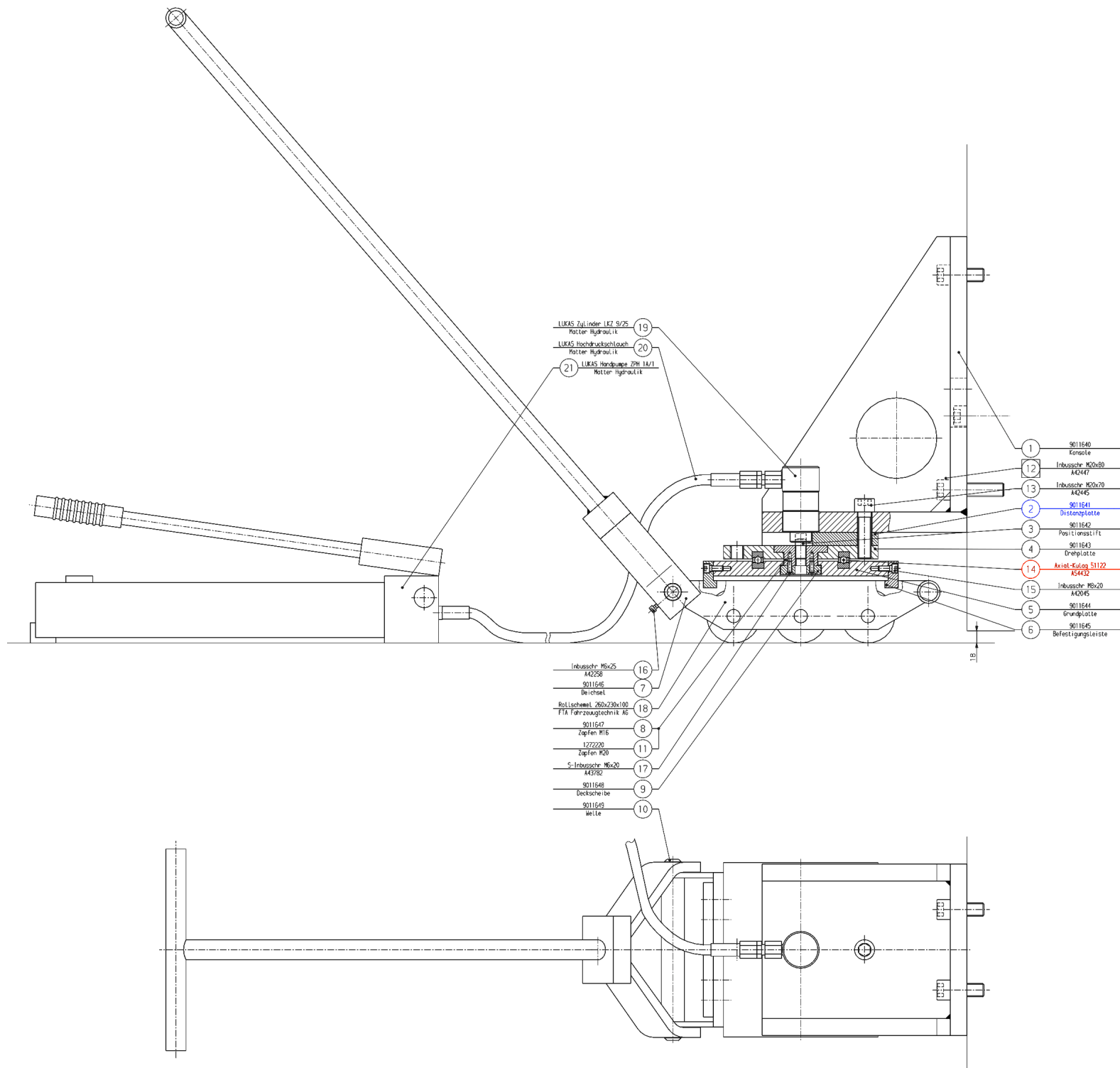
- Using pre-defined conventions (often computerized these days)
- Conventions should help facilitate reasoning
- Adding details on more accurate versions
- Postponing choices by having some open options
- Decomposing one blue print into several
- Reusing “old” blue prints (with slight changes)

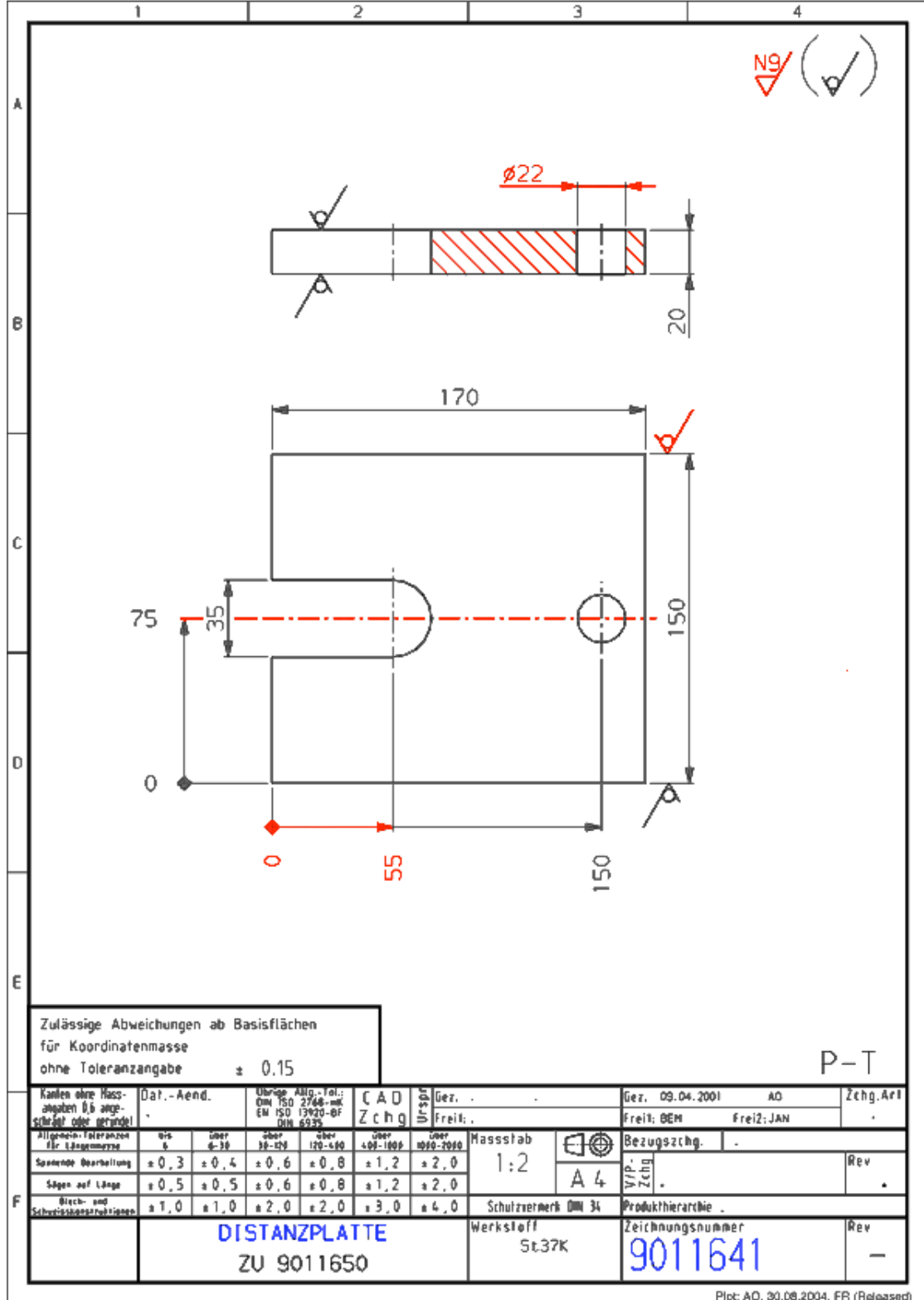
Befestigung bei guten Platzverhältnissen
(1:20)



Befestigung bei schlechten Platzverhältnissen
(1:20)







2. About the **requirement analysis** (example)

- Define main **objectives** of future system
- Define **requirements**
- Study **feasibility**

- Place of requirement document
 - System life cycle
 - Difficulties and weak point
- Role of requirement document
 - Characterizing the requirement document
 - Some structuring rules

1. Feasibility Study

2. Requirement Analysis

3. Technical Specification

4. Design

4. Coding

5. Test

6. Documentation

7. Maintenance

-
- Ensuring **relative consistency** between the phases
 - **Formal Methods** could help (in the later phases)
 - But still a problem in the **earlier phases**
 - **Weakest part:** the requirement document

- Importance of this document (due to its **position** in the life cycle)
- Obtaining a **good** requirement document is **not easy**:
 - **missing** points
 - too **specific** (over-specified)
- Requirement document are usually **difficult to exploit**
- There might exist **some guidelines** allowing us to better exploit it

- Hence **very often** necessary to **rewrite it**
- It will cost a significant amount of **time and money** (but well spent)
- The famous **specification change** syndrome might **disappear**

- Two **separate texts** in the same document:
 - **explanatory** text: the **why**
 - **reference** text: the **what**
- **Embedding** the reference text within the explanation text
- The reference text eventually becomes the **official** document
- Must be **signed** by concerned parties

2.8 The Cantor-Bernstein Theorem.

If $a \preceq b$ and $b \preceq a$ then a and b are equinumerous.

This theorem was first conjectured by Cantor in 1895 and proved by Bernstein in 1898.

Proof: Since $b \preceq a$, then a has a subset c such that $b \approx c$.

...

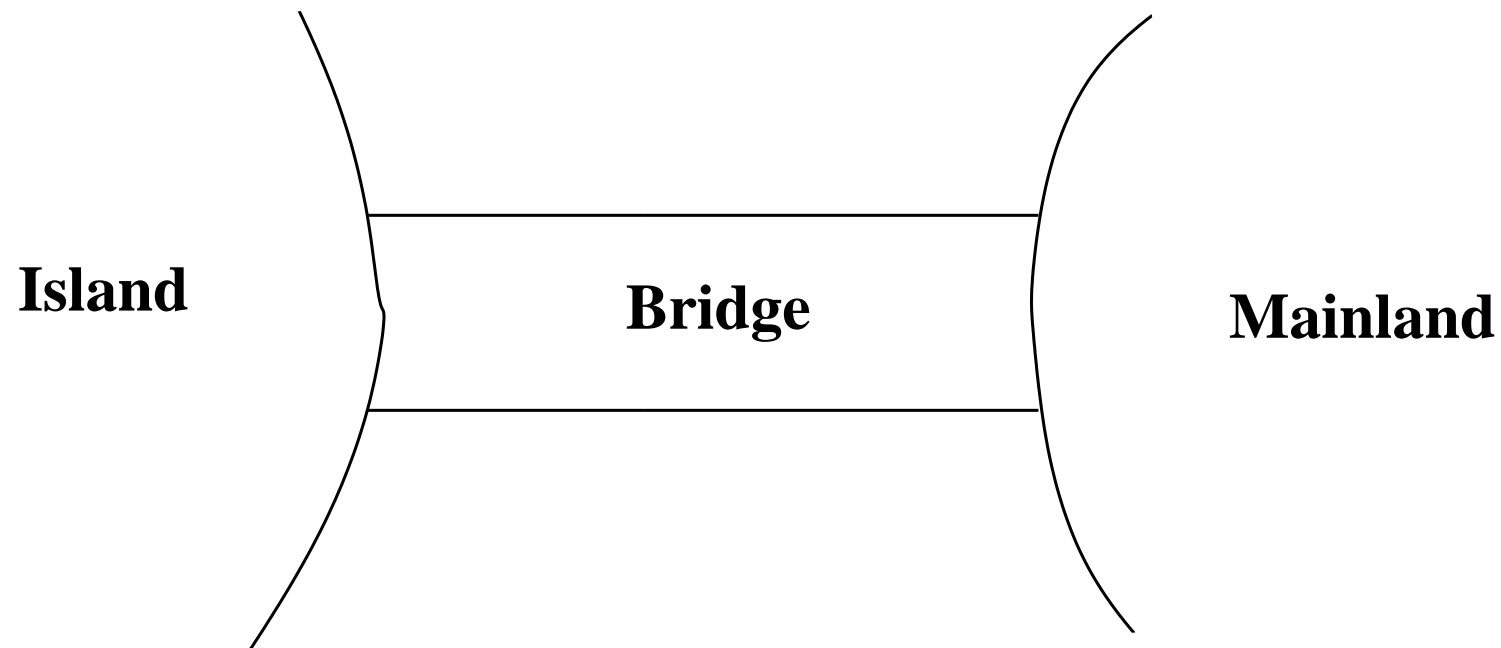


- Contains the **properties** of the future system
- Made of short **labeled** “fragments” (**traceability**)
- Should be **easy to read** (different font) and **easy to extract** (boxed)
- About the **abstraction levels** (don't care too much)
- The problem of **over-specification** (don't care too much)

- The system we are going to build is a **piece of software** connected to some **equipment**.
- There are two kinds of requirements:
 - those concerned with the equipment, labeled **EQP**,
 - those concerned with the function of the system, labeled **FUN**.
- The function of this system is to **control cars on a narrow bridge**.
- This bridge is supposed to link the **mainland** to a **small island**.

The system is controlling cars on a bridge between the mainland and an island	FUN-1
-------------------------------------------------------------------------------	-------

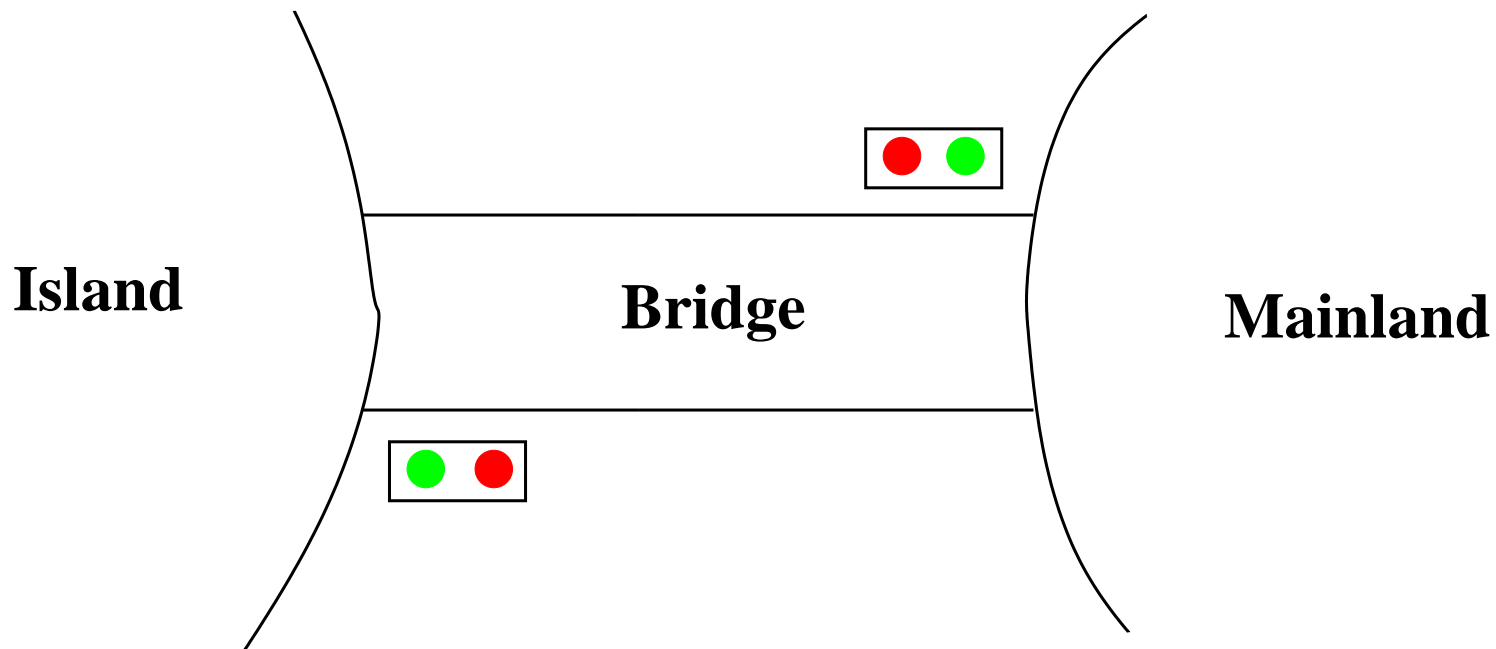
- This can be illustrated as follows



- The controller is equipped with two traffic lights with two colors.

The system has two traffic lights with two colors: green and red	EQP-1
------------------------------------------------------------------	-------

- One of the traffic lights is situated on the **mainland** and the other one on the **island**. Both are close to the bridge.
- This can be illustrated as follows:



The traffic lights control the entrance to the bridge at both ends of it

EQP-2

- **Drivers** are supposed to **obey the traffic light** by not passing when a traffic light is red.

Cars are not supposed to pass on a red traffic light, only on a green one

EQP-3

- There are also some **car sensors** situated at both ends of the bridge.
- These sensors are supposed to **detect the presence of cars** intending to enter or leave the bridge.
- There are **four** such **sensors**. Two of them are situated on the bridge and the other two are situated on the mainland and on the island.

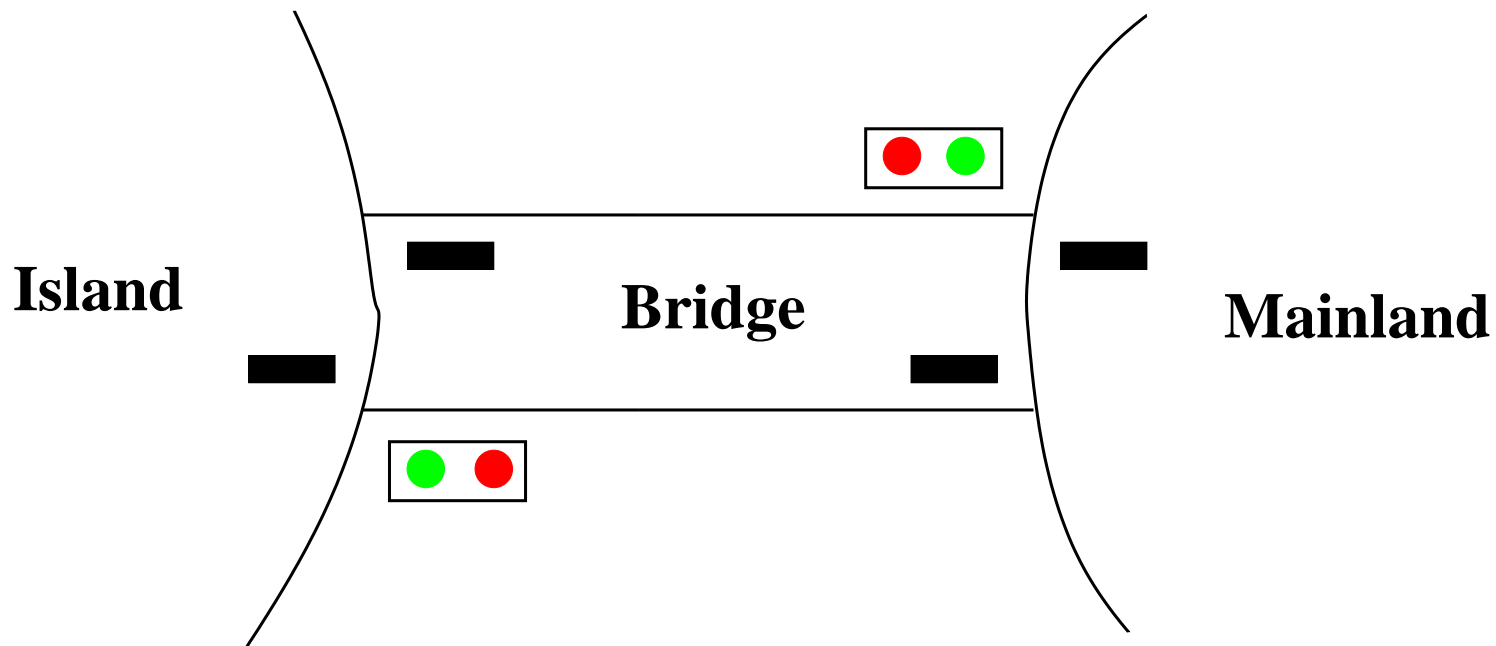
The system is equipped with four car sensors each with two states: on or off

EQP-4

The sensors are used to detect the presence of cars entering or leaving the bridge

EQP-5

- The pieces of equipment can be illustrated as follows:



- This system has two main constraints:
 - the **number of cars** on the bridge and the island is **limited**
 - the **bridge is one way**.

The number of cars on the bridge and the island is limited

FUN-2

The bridge is one way or the other, not both at the same time

FUN-3

The system is controlling cars on a bridge between the mainland and an island

FUN-1

The number of cars on the bridge and the island is limited

FUN-2

The bridge is one way or the other, not both at the same time

FUN-3

The system has two traffic lights with two colors: green and red

EQP-1

The traffic lights control the entrance to the bridge at both ends of it

EQP-2

Cars are not supposed to pass on a red traffic light, only on a green one

EQP-3

The system is equipped with four car sensors each with two states: on or off

EQP-4

The sensors are used to detect the presence of cars entering or leaving the bridge

EQP-5

- Functional
- Safety
- Equipment
- Degraded modes
- Availability
- Delays

- Short natural language statements
- Tables (“data” description)
- Transition diagrams
- Mathematical formulae
- Physical units table
- . . .

3. About modelling

- What are they used for?
- When are they to be used?
- Is UML a formal method?
- Are they needed when doing OO programming?
- What is their definition?

- Formal methods are techniques for building and studying blue prints
ADAPTED TO OUR DISCIPLINE

Our discipline is: design of hardware and software SYSTEMS

- Such blue prints are now called models
- Reminder:
 - Models allow to reason about a FUTURE system
 - The basis is lacking (hence you cannot “execute” a model)

- Reminder (cont'd):
 - Using pre-defined conventions
 - Conventions should help facilitate reasoning (more to come)
- Consequence: Using ordinary discrete mathematical conventions:
 - Classical Logic (Predicate Calculus)
 - Basic Set Theory (sets, relations and functions)
- Such conventions will be reviewed in subsequent lectures

- a “classical” piece of software
- an electronic circuit
- a file transfer protocol
- an airline booking system
- a PC operating system
- a nuclear plant controller
- a SmartCard electronic purse
- a launch vehicle flight controller
- a driverless train controller
- a mechanical press controller
- etc.

- They are made of **many parts**
- They interact with a possibly **hostile environment**
- They involve **several executing agents**
- They require a **high degree of correctness**
- Their construction spreads over **several years**
- Their specifications are subjected to **many changes**

- Fully automatic train systems
- Paris metro line 14 (October 1998)
- Roissy Airport shuttle (March 2007)
- In each case, the safety critical part only is done with this approach of Formal Method with Proofs (B)

Line length	8.5 km
Number of Stops	8
Time interval between two trains	115 s
Speed	40 km/h
Number of trains	17
Passengers per day	350,000

Line length	3.3 km
Number of Stops	5
Time interval between two trains	105 s
Speed	26 km/h
Number of trains	14
Passengers per hour	2,000

	Paris	Roissy
Number of final ADA lines (from B)	86,000	158,000
Number of proofs	27,800	43,610
Percentage of interactive proofs	8.1	3.3
Interactive proofs in Man.Month	7.1	4.6

- Man.month calculated with:
 - 15 interactive proofs per man.day
 - 21 days in a month
- In both cases, no unit tests and no integration tests
- Reinforcing global tests (catastrophic scenarios)
- Important differences in the software requirements:
 - Paris: specially done for the project
 - Roissy: adaptation from O'Hare Airport (problems)

- These systems operate in a **discrete** fashion
- Their dynamical behavior can be **abstracted** by:
 - A succession of **steady states**
 - Intermixed with **sudden jumps**
- The possible number of state changes are **enormous**
- Usually such systems **never halt**
- They are called **discrete transition systems**

- Test reasoning (a vast majority): VERIFICATION
- Blue Print reasoning (a very few): CORRECT CONSTRUCTION

- Based on laboratory execution
- Obvious incompleteness
- The oracle is usually missing
- Properties to be checked are chosen a posteriori
- Re-adapting and re-shaping after testing
- Reveals an immature technology

- Based on a **formal model**: the “blue print”
- **Gradually** describing the system with the **needed precision**
- **Relevant Properties** are chosen **a priori**
- Serious thinking made **on the model**, not on the final system
- **Reasoning is validated by proofs**
- Reveals a **mature technology**

- The proof **succeeds**
- The proof fails but **refutes the statement to prove**
 - the model is **erroneous**: it has to be modified
- The proof **fails but is probably provable**
 - the model is **badly structured**: it has to be reorganized
- The proof **fails and is probably not provable nor refutable**
 - the model is **too poor**: it has to be enriched

- Rules of Thumb:

n lines of final code implies $n/3$ proofs

95% of proofs discharged automatically

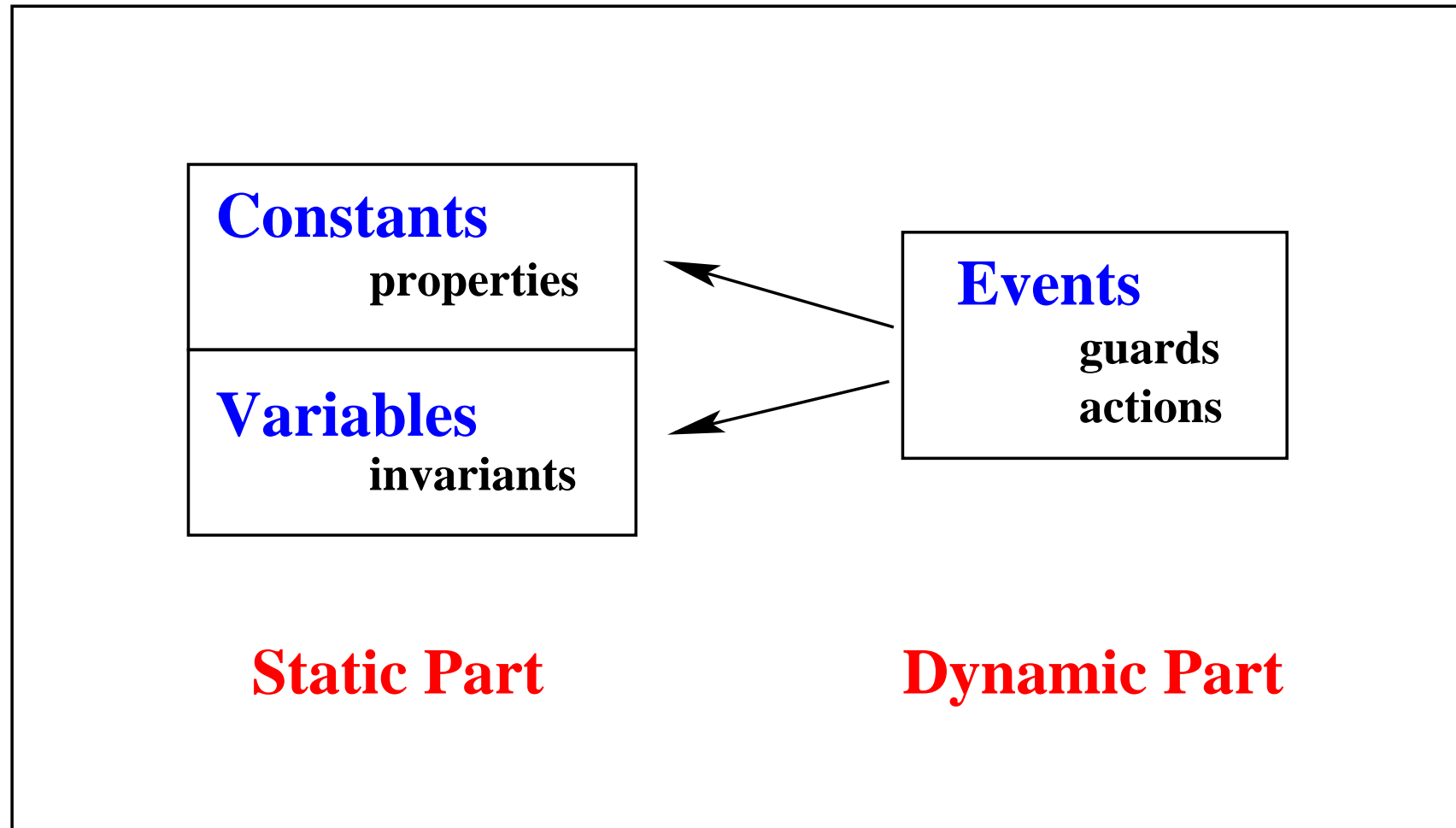
5% of proofs discharged interactively

350 interactive proofs per man-month

- 60,000 lines of final code \leadsto 20,000 proofs \leadsto 1,000 int. proofs
- 1,000 interactive proofs \leadsto $1000/350 \simeq 3$ man-months
- Far less expensive than heavy testing

- A discrete model is first made of a **state**
- The state is represented by some **constants** and **variables**
- Constants are linked by some **properties**
- Variables are linked by some **invariants**
- Properties and invariants are written using **set-theoretic expressions**

- A discrete model is also made of a number of **events**
- An event is made of a **guard** and an **action**
- The **guard** denotes the **enabling condition** of the event
- The **action** denotes the way the **state is modified** by the event
- Guards and actions are written using **set-theoretic expressions**



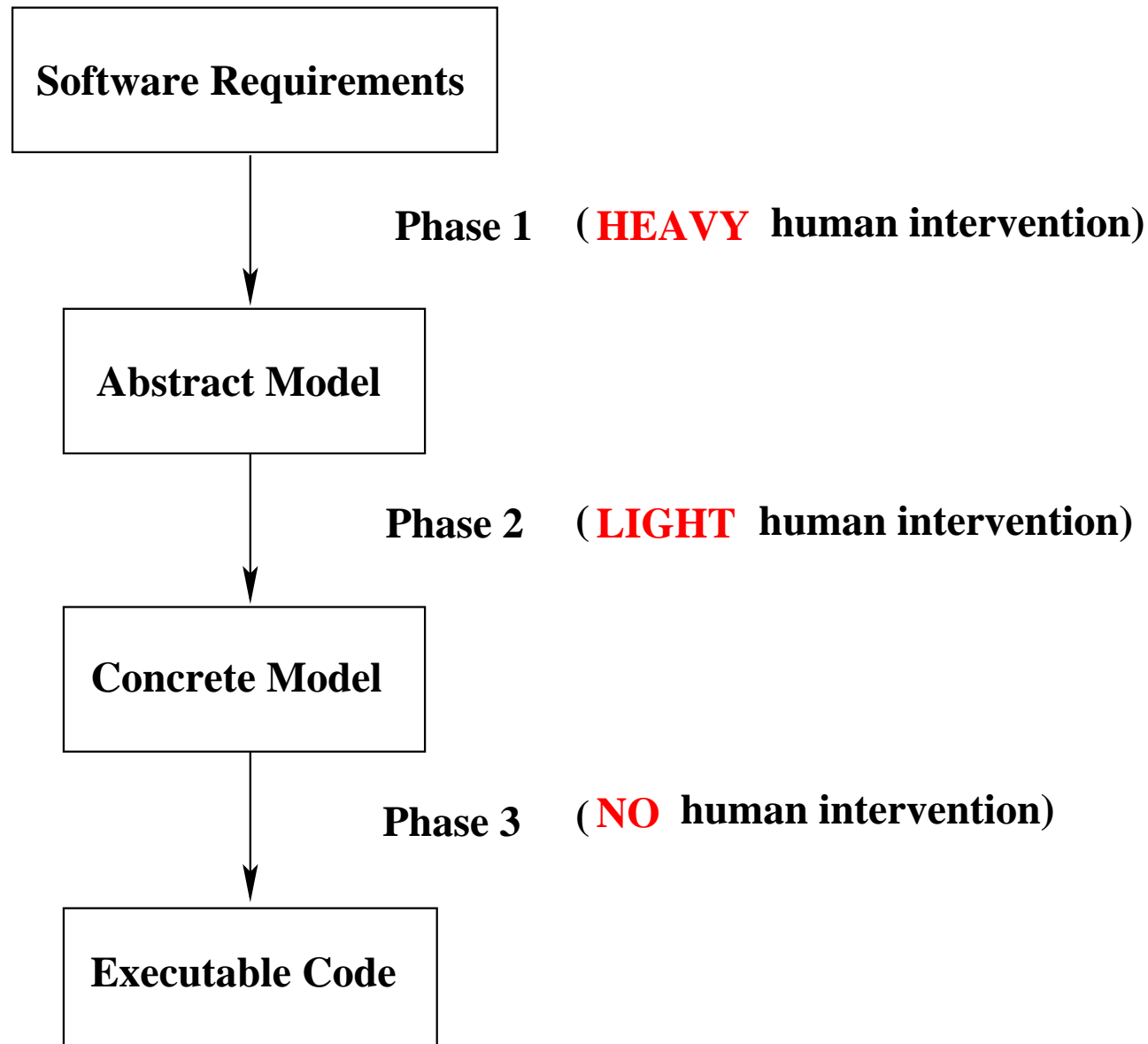
- An event execution is supposed to **take no time**
- Thus, **no two events can occur simultaneously**
- When all events have false guards, the **discrete system stops**
- When some events have true guards, **one of them** is chosen non-deterministically and **its action modifies the state**
- The previous phase is **repeated** (if possible)

```
Initialize;  
while (some events have true guards) {  
    Choose one such event;  
    Modify the state accordingly;  
}
```

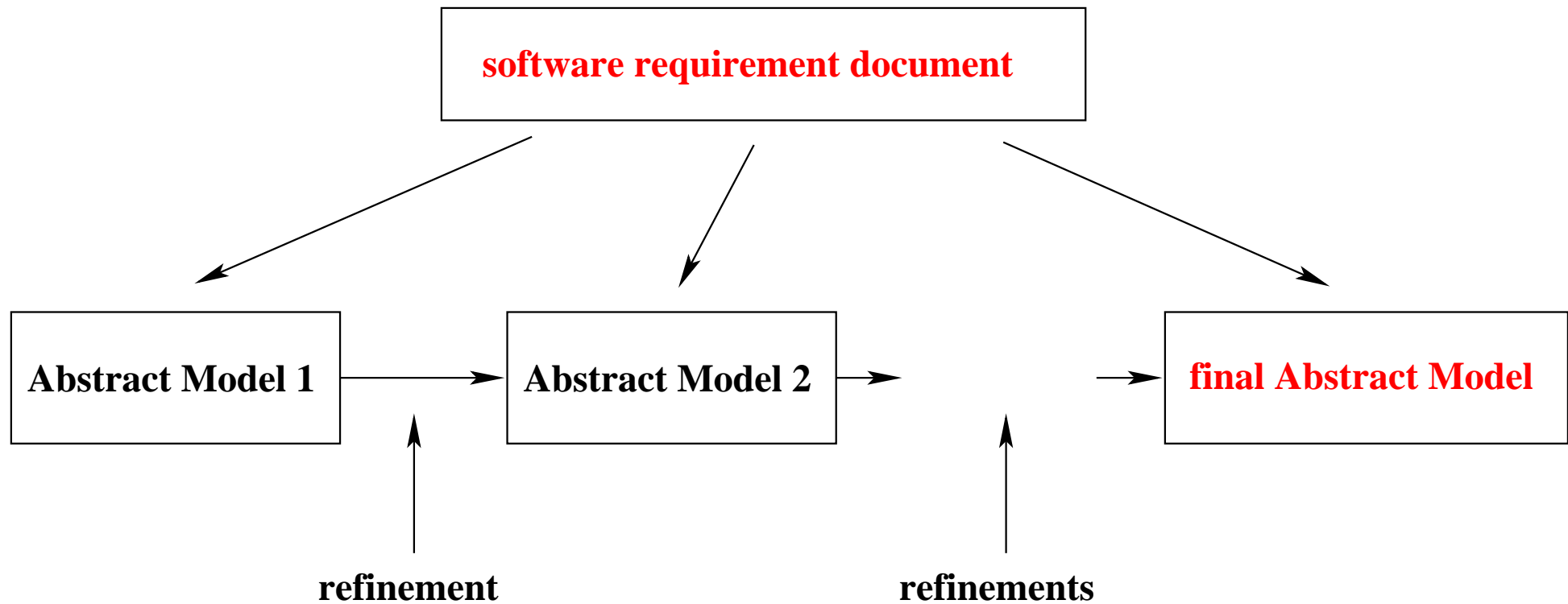
- Stopping is not necessary: a discrete system may run for ever
- This interpretation is just given here for informal understanding
- The meaning of such a discrete system will be given by the proofs which can be performed on it (next lectures)

- Formalization contains models of:
 - the **future software** components
 - the **future equipments** surrounding these components
- The overall **model construction** can be **very complex**
- Three techniques can be used to master this complexity
 - **refinement**
 - **decomposition**
 - **generic instantiation**

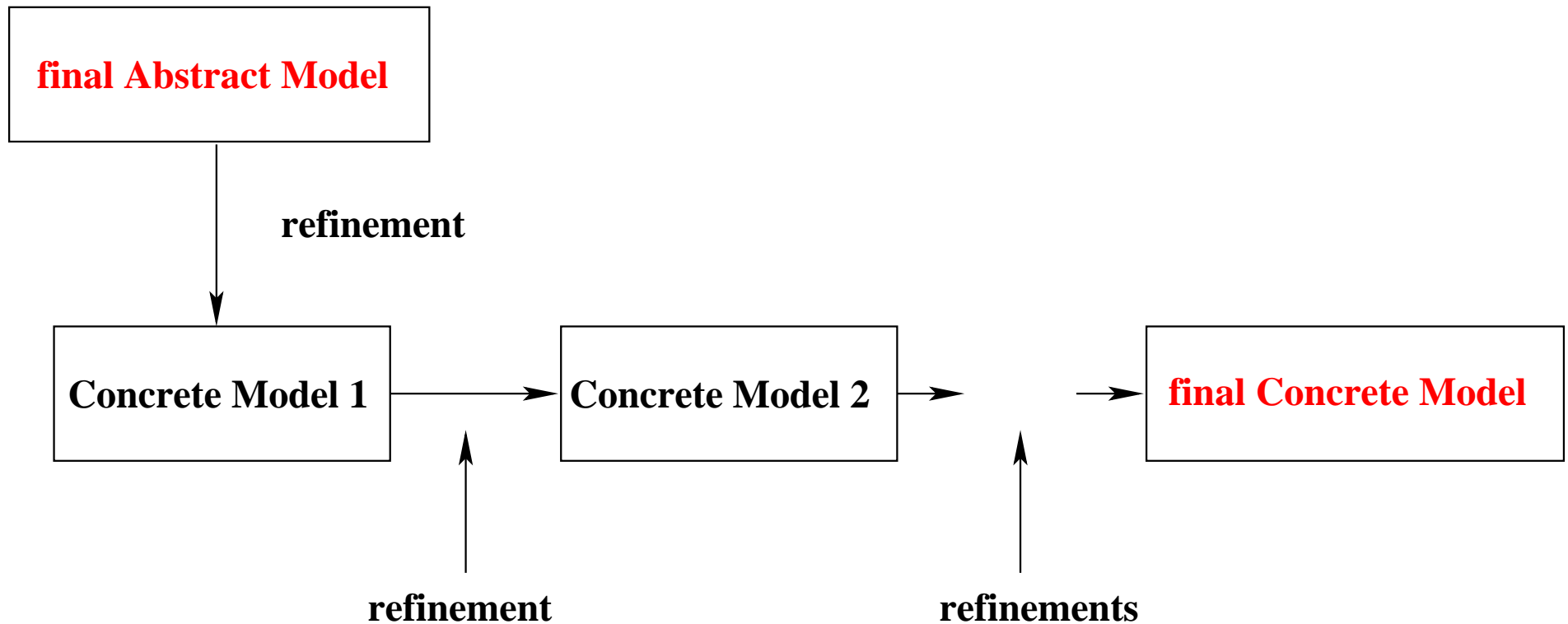
-
- Refinement allows us to build model **gradually**
 - We shall build an **ordered sequence** of more precise models
 - Each model is a **refinement** of the one preceding it
 - A useful analogy: looking through a **microscope**
 - **Spatial** as well as **temporal** extensions
 - **Data refinement**



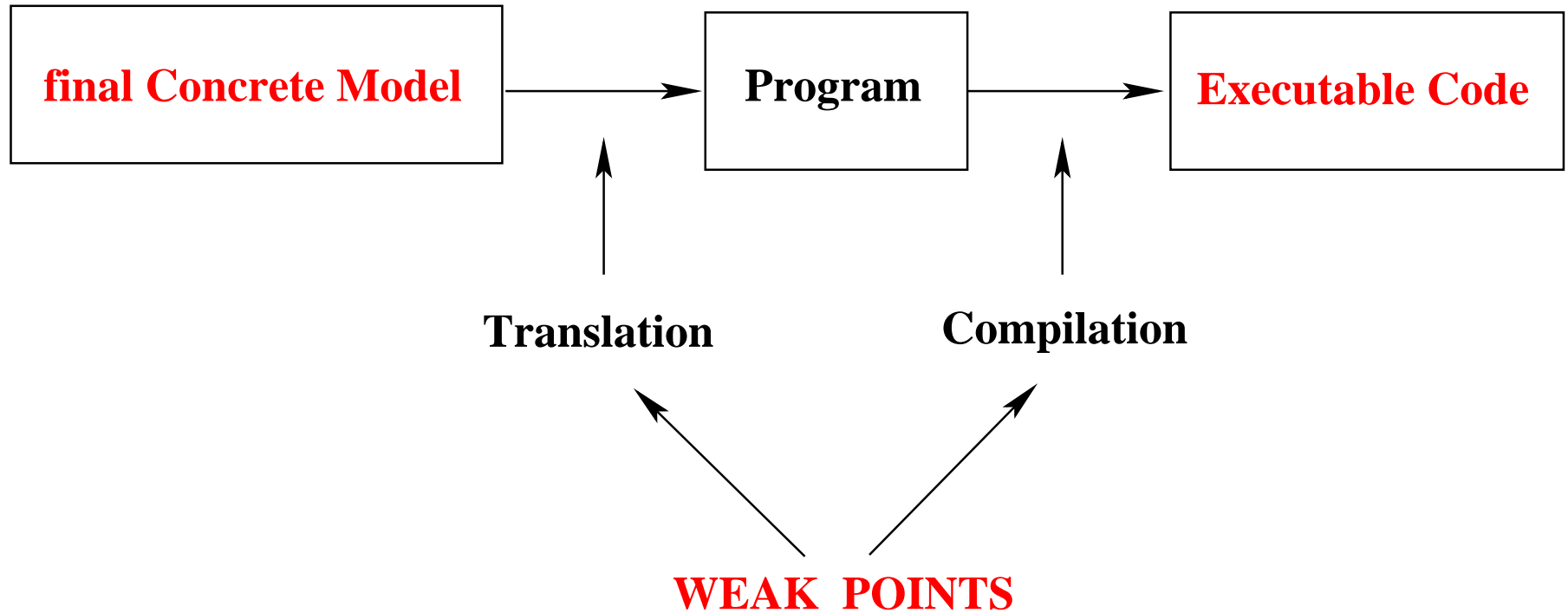
- The **software requirement document** is given
- **Details** of this document are **gradually extracted**
- The Abstract Model is thus constructed by **successive refinements**
- Each refinement is an **independent model**



- The point of departure is now the **final abstract model**
- The concrete model construction is done in a **similar manner**:
by **successive refinements**
- the set-theoretic constructs are **gradually transformed** into **computerizable objects**: finite scalars, arrays, pointers, etc.
- The non-deterministic operations are **gradually transformed** into **programming constructs**: conditional, loops, procedure calls, etc.

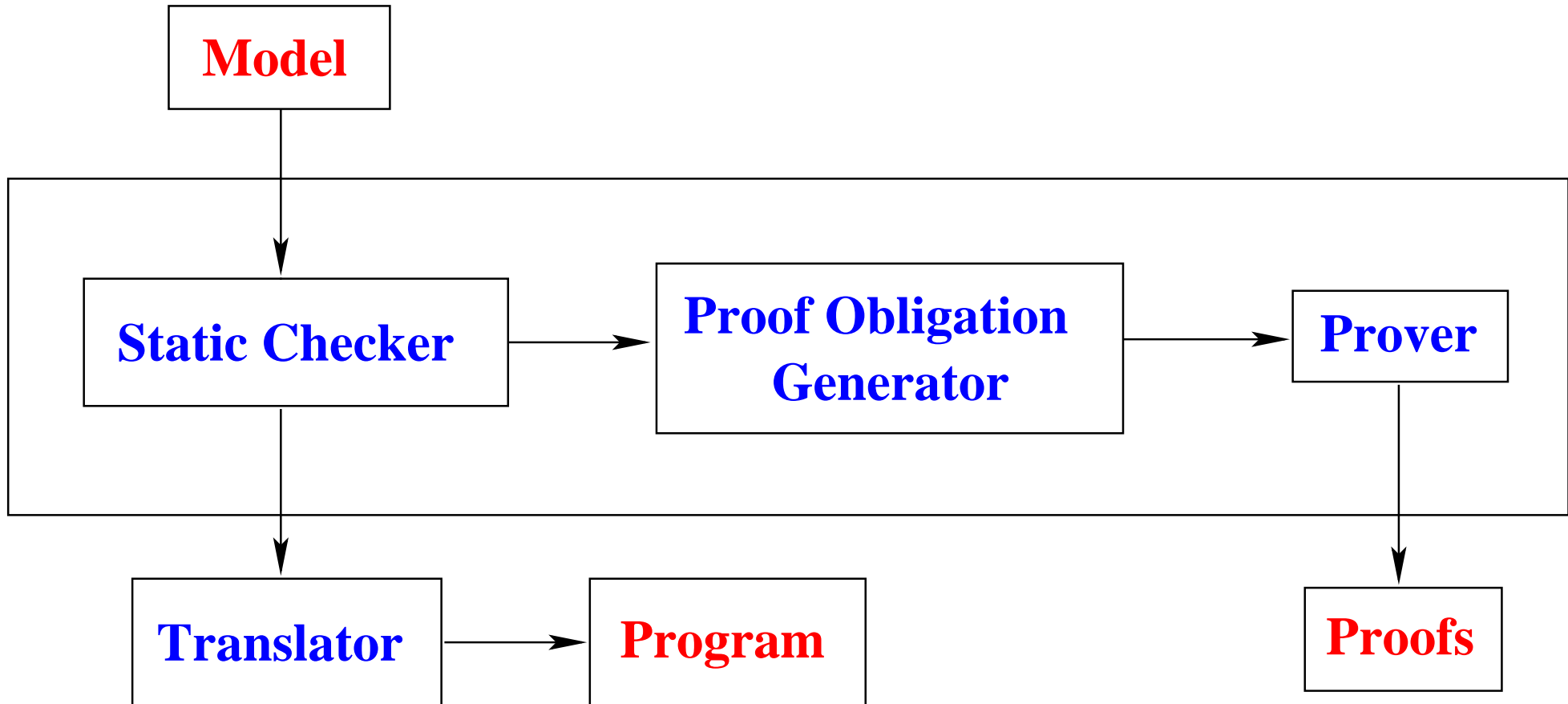


- The Point of Departure is now the **final concrete model**
- The concrete model is **checked by a translatability tool**
- The concrete model is **translated into a classical program**
- The classical program is **compiled into executable code**
- Translation and compilation are **weak points** (**more later**)



- Three phases:
 - Constructing the **abstract model**
 - Constructing the **concrete model**
 - Constructing the **executable code**
- Two main concepts:
 - **Refinement**
 - **Proof**

- The **Static Checker**
- The **Proof Obligation Generator**
- The **Prover**
- The **Translator** from B to a programming language



- Refinement does not solve **alone** the problem of complexity
- Models can become very large: **large state and many events**
- A large model can be **decomposed** into smaller ones
- Decomposed models **communicates**
- Decomposed models can then be **refined further** and so on

- Models can be parameterized with **carrier sets** and **constants**
- Constants are specified by means of **properties**
- Analogy with **algebraic theories** (i.e. group theory)
- Such models are said to be **generic**
- They can be **instantiated**